Correctness Proof of a Middleware GSI Certification-Based Replication Protocol

J.R. González de Mendívil, J.E. Armendáriz, F.D. Muñoz, J.R. Garitagoitia

Dpto. Ing. Matemática e Informática - Univ. Pública de Navarra - Pamplona (Spain)
Instituto Tecnológico de Informática - Valencia (Spain)

{mendivil,enrique.armendariz,joserra}@unavarra.es, fmunyoz@iti.upv.es

Technical Report ITI-ITE-07/17

Correctness Proof of a Middleware GSI Certification-Based Replication Protocol

J.R. González de Mendívil, J.E. Armendáriz, F.D. Muñoz, J.R. Garitagoitia

Dpto. Ing. Matemática e Informática - Univ. Pública de Navarra - Pamplona (Spain)
Instituto Tecnológico de Informática - Valencia (Spain)

Technical Report ITI-ITE-07/17

e-mail: {mendivil,enrique.armendariz,joserra}@unavarra.es, fmunyoz@iti.upv.es

December 14, 2007

Abstract

This paper provides a formal specification and proof of correctness of a basic *Generalized* SI certification-based data replication protocol for database middleware architectures. It has been modeled using a state transition system, as well as the main system components, allowing a perfect match with the usual deployment in a middleware system. The proof encompasses both safety and liveness properties, as it is commonly done for a distributed algorithm. This approach enables the analysis of multiple specific variations on this basic protocol, that have been included in previous works. Furthermore, a crash failure model has been assumed for the correctness proof, although recovery analysis is not the aim of this paper, this allows an easy extension towards a crash-recovery model support in future works. Notice that most of previous works have focused in the safety part, here it is considered the liveness part too, in particular, the uniform commit: if a site has committed a transaction the rest of sites will either commit it or have crashed.

1 Introduction

Replication is a common technique for improving the availability of both data and processes. Data and their associated processes are very important for many kinds of enterprise services. So, database replication has received a lot of attention in the academical field for years [47, 11, 2, 37, 49, 26, 35] and also by the *database system* (DBS) companies [34, 48] although the latter tend to use more conservative and safer strategies.

Recent papers [50] have compared the performance of different database replication protocols and have shown that those based on a certification strategy and total-order broadcast of updates performed by a transaction, featured by a Group Communication System (GCS), provide the best performance (i.e., minimal response time) in most system configurations. Certification-based protocols rely on the local execution of the transaction operations on its delegate replica. When the client application requests the transaction commit, its updates (denoted as writeset) and its read operations (respectively, readset) are collected in two different sets and multicast in total-order to all replicas. Once such sets are delivered, they are certified against a historic list of previously committed transactions and, if no conflict arises, the transaction is applied and committed in each replica. Otherwise, the transaction is aborted in the delegate replica and discarded in the other ones. Notice that the certification process is symmetric and can be independently executed in each replica, providing the same results. Due to this, no additional voting phase is needed to decide whether a transaction should commit or abort.

Despite their short transaction completion time, certification-based replication protocols are not a clear best option for managing transactions in the serializable isolation level, since readsets need to be collected and propagated in such level. However, things are quite better in the *snapshot isolation* [7] (SI, for short) level: readsets are not needed in the certification process when a multi-version concurrency control mechanism is used by the underlying DBS. Although SI is more relaxed than serializable there have been some works [18] that describe how to ensure serializable executions using the SI level. So, multiple replication protocols [26, 51, 30, 17, 33] have been published using such combination: a certification-based protocol providing the snapshot isolation level. There are good reasons for this: besides a shorter completion time (since readsets are not collected, nor transferred, nor evaluated in the certification process), the isolation achieved is almost as strict as in the serializable case, and read operations are never blocked. Additionally, when a *Generalized SI* (GSI) level is used, as proposed by [17] and followed by most other academical works on SI, transaction start does not need to block [19] further improving the protocol performance.

In order to improve their portability and DBS independence, many replication protocols [43, 3, 12, 39, 5, 30, 35, 22, 33] have been implemented in a middleware [9] layer. This penalizes their performance and demands some additional support at the middleware layer for managing several protocol details that could have been easily implemented at the DBS core (concurrency control, readset and writeset collection, . . .). This is partially overcome with the help of standard mechanism for reusing the underlying DBS support at the middleware layer, as [33] describes for writeset conflict detection, and by the middleware enhanced portability. Thus, without many efforts, the middleware system can work in a heterogeneous environment where different DBSs are being used in the system nodes.

Actually, most of replication protocols [33, 30, 17, 51, 23, 14, 40] that provide GSI, or other different SI flavors, for replicated environments are certification based [33, 30, 17, 51] with several enhancements to increase their performance, e.g. concurrent execution of disjoint writesets as in [30], that do not carefully pay attention to maintain their correctness. Parallel to this, most of them ensure that they can afford replica failures but, to the best of our knowledge, lack of any correctness proof about this fact; even for the most intuitive and simple scenario such as the crash failure. Due to these two important aspects, in this paper we propose a formal specification and correctness proof of a basic certification-based replication protocol providing GSI. This has been done by way of a state transition system, as presented in [46]. This formalism has permitted us to represent independently each component involved in the middleware database replication system (i.e. the DBS, the GCS and the Replication Protocol (RP) modules) and the interaction and composition of their associated events. For instance, this assures that the interaction between the replication protocol and the underlying DBS only uses a few standard operations, and that no core-dependent DBS facility is available to the replication protocol.

In this work, it has been followed the traditional approach on identifying and verifying safety and liveness properties for a distributed system. Moreover, there are some additional practical issues that must be considered for ensuring liveness properties such as in the case of application of an already certified writeset. Under this scenario, this writeset must be committed and this will not be possible unless the RP does not take some control over transactions already being executed in the DBS. We are not aware of previously research works that have taken these side-effects into account.

RP follows the basic certification-based approach [50]; no further optimization has been considered since our aim is to emphasize several correctness details throughout all the work. A transaction is firstly executed at its delegate replica (that transaction is denoted as local) and, thus, obtains its latest snapshot version which does not necessarily be the latest system version. When the commit operation is requested, its associated writeset is sent, along with its snapshot version, to all available replicas using the total-order broadcast [13]. Upon the delivery of the message, each server executes a deterministic certification phase that decides the final outcome of the transaction (either aborted or committed). In the case of GSI [17], RP maintains the sequence of already certified transactions that lets it to determine whether it intersects with any of the writesets of transactions with version numbers higher than the snapshot gotten by the delivered writeset. If so, the transaction needs to be aborted. Otherwise, all intersections are empty, the writeset can be committed and is applied as a remote transaction (it only needs to be committed at its delegate).

Thus, the contributions of this paper are: (a) formal specification of a basic certification-based GSI data replication protocol along with the rest of components of a middleware architecture (such as the DBS and the GCS), (b) its formal correctness proof assuming a crash failure model, distinguishing safety and liveness properties as in any other distributed algorithm, (c) to provide the basis –using the two previous issues– for studying several variations of this basic protocol that enhance its performance without incurring in the violation of its correctness, and, (d) to ensure that the formalization given is appropriate for a middleware architecture.

The rest of this paper has been structured as follows. Section 2 describes the system model and the formalism being used in the protocol specification. Since a middleware architecture is being assumed and such middleware needs some underlying services, Sections 3 and 4 describe the two system components providing such services: the DBS and the GCS, respectively. Later, Section 5 specifies and comments the proposed replication protocol. Such protocol is proven correct in Section 7, some related work is described in Section 8, and the paper is concluded in Section 9.

2 System Model and Presentation Formalism

The system (Figure 1) considered in this paper is an abstraction of a middleware database replication architecture. The system is composed by $|N|$ sites (or nodes), being N the set of site identifiers, which communicate among them using a GCS [13].

Each site $n \in N$ contains a Database System (DBS_n) including a copy of the entire database schema. We assume a fully replicated system where each site runs a Replication Protocol (RP). Figure 1 describes how components interact with each other at a site. T is the set of transaction identifiers. Each transaction $t \in T$ contains a unique identifier that points out the site where the transaction is firstly started (its delegate replica) denoted as $t.site \in N$.

We distinguish among three different components in our system. Besides the underlying DBS_n each site $n \in N$ has a copy of RP being used and has also access to the system-wide GCS. Note that transactions are started by client applications, but the execution flow of a transaction is managed by the RP component and no user application component is distinguished. So, client actions are delegated on the RP and the latter is the agent that starts transactions in the DBS_n component. Modeling client applications introduces no benefit, since our aim is to prove that the RP actions comply with the correctness criteria of a certification-based replication protocol, and other client actions are not relevant for this.

In this paper, we will assume that databases provide SI [7]. Regarding failures, we assume no Byzantine failures occur; i.e., sites do not behave in a malicious manner. A site behaves according to its specification until it possibly crashes. After a crash event components of a site stop their activity. We assume a *partially synchronous distributed system* in which up to f sites may fail; i.e. there is at least $|N| - f$ correct sites. A site is *correct* if it never fails, in other case it is a *faulty* site. The set $Correct(N)$ holds all correct sites in the system. This set does not necessarily be the same or known in advance, it merely simplifies the way certain properties are specified.

We also assume that the GCS is able to guarantee the *atomic broadcast* [21] communication primitive with the additional property of preventing the contamination phenomenon [15]. Informally, this means that if a site delivers a message m' after m then another site delivers m' only after m has been delivered. Therefore, there are no gaps in the delivery process at any site.

The RP coordinates the execution of transactions among different sites to ensure the replica consistency and to guarantee the GSI level [17]. In the next subsection, we overview the formal framework used to specify components and to define the RP.

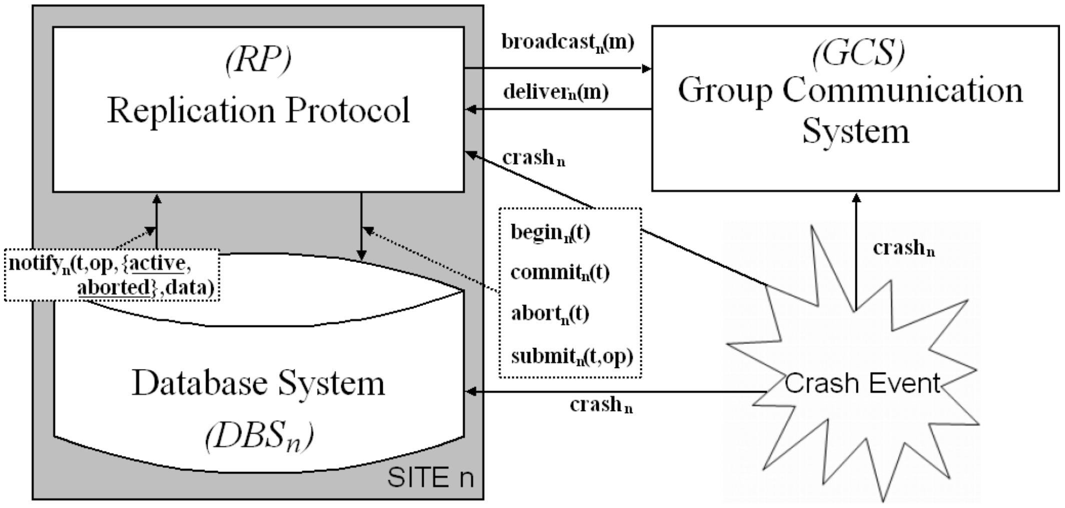

Figure 1: Main components of the system.

2.1 The Formal Framework

Figure 1 shows our system as a set of interacting components. Our formalization is based on the work of Shankar [46], where a distributed system is modeled using a state transition system and properties can be proven on the executions of the system. Although such paper does not discuss any composition model, it suggests a composition approach that is followed in this document. In a nutshell, our RP is a state transition system which models the composition of all the RP instances in each site. Each RP instance at each site $n \in N$ interacts with its local DBS_n and with the globally shared GCS. Since this work is not interested on the implementation details of the DBS_n and the GCS components, only the relevant properties of these components have been considered.

In the following, we introduce the framework used here.

Component Specification. In order to model the specification of a system component, C, we give its *external interface* and a collection of *trace* properties. The external interface of C, denoted $Events(C)$, defines the possible events the component may engage. A trace is a finite or infinite sequence of events where each event belongs to $Events(C)$. The set of traces of C is denoted as $Traces(C)$. A finite trace is denoted $\beta = \pi_1\pi_2 \dots \pi_j$, an infinite trace as $\beta = \pi_1\pi_2...\pi_j...$; and a prefix of length j, being $|\beta| \geq j \geq 0$, of a trace β by $\beta[j]$. Properties over traces are modeled as assumptions. The component satisfies its specification if each possible trace verifies the set of defined assumptions.

State Transition System. We now outline the model in [46] based on state transitions systems for concurrent and distributed systems. A state transition system C is defined by:

- $Variables(C)$. A set of variables and their domains.
- $Initial(C)$. An initial condition on $Variables(C)$.
- $Events(C)$. A set of events.
- For each event $\pi \in Events(C)$:
  - $enabled(\pi)$, an enabling condition (a predicate in $Variables(C)$); and
  - $action(\pi)$, an action (sequential program that updates $Variables(C)$);
- A finite description of fairness requirements.

Each possible value assignment to $Variables(C)$ defines a particular state of the transition system C. Thus, the set of states of C is the cartesian product of the variable domains. Part of these states are initial configurations and they are defined by $Initial(C)$. We assume that the set of initial states is non-empty. For each event π, its associated enabling condition, $enabled(\pi)$, and action, $action(\pi)$, define a set of *state transitions*. More formally, the set of state transitions is defined as $\{(p, \pi, q) \colon p, q$ are system states; p satisfies $enabled(\pi)$; and q is the result of executing $action(\pi)$ in $p\}$. For each event π, $action(\pi)$ is executed atomically and always terminates.

An *execution* is a sequence of the form: $\alpha = s_0\pi_1s_1 \dots \pi_zs_z \dots$ where the s_z's are system states, the π_z's are events, s_0 is an initial state, and every (s_{z-1}, π_z, s_z) is a transition of π_z. The content of a state variable, var, at a given system state, s_z, is denoted as $s_z.var$. If the $action(\pi_z)$ does not modify a given state variable var', it will keep its previous value ($s_z.var' = s_{z-1}.var'$). An execution can be infinite or finite. By definition, a finite execution ends in a state. The final state of a finite execution is a *reachable* state. Note that for any execution α, every finite prefix of α ending in a state is also an execution. A finite execution α ending in the reachable state s_z will be denoted as $\alpha[z]$.

In the following, we assume that each event is weak-fair. Informally, it means that if an event π is enabled continuously in an execution, then it eventually occurs (see [46] for a more formal definition). Thus, a fair execution of C is an execution verifying the fairness requirement of C. The set of all possible fair executions of C is sufficient for defining its liveness and safety properties.

Finally, as we are describing a distributed system, we use a subscript for each state variable and event to denote where the state variable belongs to and in which site the event is executed, respectively.

Component Interaction. A state transition system C is able to interact with other component C' via executing an event $\pi' \in Events(C')$ of the component as part of effects of an $action(\pi)$ being $\pi \in Events(C)$. In that case, we require that π' be non-blocking in order to guarantee the termination of $action(\pi)$. Thus, the event π' is simply "a call" from C's point of view. In the same way, the component C' is able to interact with a state transition system C via executing an event $\pi' \in Events(C')$ which is also an event of C, $\pi' \in Events(C)$. In this case, it is required that $enabled(\pi') \equiv \texttt{true}$ in C. So, the event π' of C can be considered an "upcall" from C''s point of view.

Composition. Let C be a state transition system and C' a component that is used by C. The complete system is formed by (C, C'). If C and C' follow the component interaction rules previously introduced, then for each execution α of C the following sequence $\beta(\alpha) \in Traces(C')$ can be associated to each α. The $\beta(\alpha)$ is built from α in the following manner: by collecting the events corresponding to C' from the corresponding action and events of α in the same order they are executed in α. In other words, let $\pi \in Events(C)$ and part of the effects of $action(\pi)$ include the execution of a (actually, ordered) set of events $\{\nu_i \colon \nu_i \in Events(C')\}$ then for each occurrence of π in α the sequence of $\beta(\alpha)$ is appended with $\{\nu_i \colon \nu_i \in Events(C')\}$ in the very same order they are invoked. It is worth noting that if s_z is a reachable state of α we have that $\alpha[z] \preceq$[1] α and $\beta(\alpha[z]) \preceq \beta(\alpha)$. As a concluding remark, the properties verified by C' upcalls can be used in α in order to prove the safety and liveness properties of the whole system.

3 Database System Specification

We assume that each site $n \in N$ in the distributed system stores a copy of the database DB_n which contains a collection of uniquely identified items, $items(DB_n)$. The same data item, $X \in items(DB_n)$, may have several different versions in the database. A transaction $t \in T$ creates a version X_t of data item X by performing a write operation on it, this version will be installed when the transaction is committed and $X_t \in DB_n$. This database is managed by a database system (DBS_n). In this case, at any time the current *database snapshot* includes only the latest committed versions of all data items until that time. A transaction will read objects from the snapshot gotten when it began (including its own updates), i.e. if t reads data item X it reads the version gotten by its snapshot $X_{t'} \in DB_n$ with t' the latest transaction that wrote on X before t started. At each site $n \in N$ the RP instance running at it interacts with the DBS_n using the interface shown in Figure 1, namely to execute transactions. The DBS_n supports the concurrent execution of these transactions running under SI [7] level. In what follows, we provide the details of the DBS_n behavior.

A transaction, $t \in T$, is a sequence of operations, $op \in OP$, on database items ended by a commit or abort operation. Each non-final operation may be of type read or write, $type(op) \in \{\texttt{read}, \texttt{write}\}$, and may access to a set of data items, $items(op) \subseteq items(DB_n)$. Each transaction, t, starts with a $\mathbf{begin}_n(t)$ initial action. After that, the transaction may submit an operation, op, using $\mathbf{submit}_n(t, op)$ event. Furthermore, inside each transaction, a new operation can only be submitted once the previous one has been terminated; in our case by the $\mathbf{notify}_n(t, op, result, data)$ event. The parameter $result$ contains the final effect of the operation and the transaction can be, in terms of its status, either $active$ or $aborted$. In the first case, the transaction may go on in its activity; otherwise, it has been aborted by the database which corresponds to, e.g., a transaction that does not fulfill the isolation level requirements. The parameter $data$ contains the versions of the database items read by the transaction, we will use it when necessary (i.e. read operations). A transaction t can be aborted by the RP at any time using the $\mathbf{abort}_n(t)$ event. In our model, we assume that if the last operation of a transaction has a $result = active$ the transaction can be committed at any time by the $\mathbf{commit}_n(t)$ event. The RP at each site sees these actions as the primitives to access the local database. The events $\mathbf{begin}_n(t)$, $\mathbf{submit}_n(t, op)$, $\mathbf{commit}_n(t)$ and $\mathbf{abort}_n(t)$ invoked by the RP instance at $n \in N$ always terminate. The action $\mathbf{notify}_n(t, op, result, data)$ is an upcall executed by the DBS_n upon the receipt of a $result$ for the operation op of t. Finally, the $\mathbf{crash}_n$ event models the failure of the DBS_n component.

We can think of each operation $op \in OP$ as equivalent to a single SQL statement. We can assume that multiple consecutive SQL statements can be logically integrated into a single $\mathbf{submit}_n(t, op)$. The notification event for this group of statements, via $\mathbf{notify}_n(t, op, result, data)$, will be $active$ if all of their respective sentences were successfully applied; and, otherwise $aborted$.

Therefore, at each site $n \in N$, the DBS_n has the following set of events, $Events(DBS_n) =$

$$\{\mathbf{begin}_n(t), \mathbf{commit}_n(t), \mathbf{abort}_n(t) \mid t \in T\} \cup$$
$$\{\mathbf{submit}_n(t, op) \mid t \in T, op \in OP\} \cup$$
$$\{\mathbf{notify}_n(t, op, result, data) \mid t \in T, op \in OP, result \in \{active, aborted\}, data \subseteq DB_n\} \cup$$
$$\{\mathbf{crash}_n\}$$

The traces of the DBS_n are finite or infinite sequences of events from $Events(DBS_n)$. The set of all possible traces is denoted as $Traces(DBS_n)$. One can note that the first parameter of each event different to $\mathbf{crash}_n$ is a transaction t in T. We define the function $tran \colon Events(DBS_n) \to T$ which returns the transaction which an event makes reference to. In the following, we show what assumptions traces verify in order to satisfy the specification of the DBS_n. Every assumption takes a trace, $\beta_n \in Traces(DBS_n)$, as a parameter, and we suppose it is implicit. The quantification of some of the free variables in the assumptions are clear from its context. In the next assumption we state the well-formed behaviors of traces and use $prev\_event(i, j, t)$ as a predicate which is true iff ν_i is the immediate previous event before ν_j for transaction t in a trace β_n[2].

[1]The $\preceq$ symbol stands for prefix order.

[2]$\beta_n \colon prev\_event(i, j, t) \equiv i < j \wedge tran(\nu_i) = tran(\nu_j) = t \wedge \nexists k, i < k < j \colon tran(\nu_k) = t$

Assumption 1 (Well-formed traces)**.** *Let* $\beta_n = \nu_1\nu_2\ldots\nu_p\ldots$ *and* $\beta_n \in Traces(DBS_n)$*, then* β_n *will be a well-formed trace if it satisfies the next:*

1. $tran(\nu_i) \in T \Rightarrow \nexists\, k, k < i \colon \nu_k = \mathbf{crash}_n$
2. $\nu_i = \mathbf{begin}_n(t) \Rightarrow \forall\, k, k < i \colon t \neq tran(\nu_k)$
3. $\nu_i \in \{\mathbf{commit}_n(t), \mathbf{abort}_n(t), \mathbf{notify}_n(t, op, aborted) \mid op \in OP\} \Rightarrow \forall\, k, k > i \colon t \neq tran(\nu_k)$
4. $\nu_j = \mathbf{submit}_n(t, op) \Rightarrow prev\_event(i, j, t) \wedge \nu_i \in \{\mathbf{notify}_n(t, op', active), \mathbf{begin}_n(t) \mid op' \in OP\}$
5. $\nu_j = \mathbf{notify}_n(t, op, result) \Rightarrow prev\_event(i, j, t) \wedge \nu_i = \mathbf{submit}_n(t, op)$
6. $\nu_j = \mathbf{commit}_n(t) \Rightarrow prev\_event(i, j, t) \wedge \nu_i \in \{\mathbf{notify}_n(t, op, active) \mid op \in OP\}$

In the previous Assumption (1.1) reflects the fact that after a $\mathbf{crash}_n$ event the DBS_n stops its activity; (1.2) indicates that the first event of a transaction t is $\mathbf{begin}_n(t)$; (1.3) states that after commit or abort there is no event for t; (1.4) indicates that an operation can be submitted if the result of the previous one is $active$ or the transaction is at the beginning; (1.5) states that a notification of an operation follows its submission; and, (1.6) reflects the main fact that the $\mathbf{commit}_n(t)$ event only can be provided if the transaction is active.

Notice that after a $\mathbf{submit}_n(t, op)$ the transaction gets blocked until its associated $\mathbf{notify}_n(t, op, active)$ event happens. The next Definition is introduced in order to capture the different states a transaction may switch ($active$, $blocked$, $committed$ and $aborted$). In the definition, $T(\beta_n)$ refers to transactions in T such that they have started in β_n; i.e. $t \in T(\beta_n)$ iff there exists the event $\mathbf{begin}_n(t)$ in β_n.

Definition 1 (Transaction states)**.** *Let* β_n *be a trace of* $Traces(DBS_n)$*. For every* $\beta_n[j]$*,* $0 \leq j \leq |\beta_n|$*, the transaction states of the transaction* $t \in T$ *are defined as follows:*

1. $idle_n(t, j) \equiv (t \in T \setminus T(\beta_n[j]))$
2. $active_n(t, j) \equiv \exists\, i, i \leq j \colon (\nu_i \in \{\mathbf{notify}_n(t, op, active), \mathbf{begin}_n(t) \mid op \in OP\} \wedge \forall\, k, i < k \leq j \colon t \neq tran(\nu_k))$
3. $blocked_n(t, op, j) \equiv \exists\, i, i \leq j \colon (\nu_i = \mathbf{submit}_n(t, op) \wedge \forall\, k, i < k \leq j \colon t \neq tran(\nu_k))$
4. $committed_n(t, j) \equiv \exists\, i, i \leq j \colon \nu_i = \mathbf{commit}_n(t)$
5. $aborted_n(t, j) \equiv \exists\, i, i \leq j \colon (\nu_i \in \{\mathbf{notify}_n(t, op, abort), \mathbf{abort}_n(t)\} \vee (\nu_i = \mathbf{crash}_n \wedge (blocked_n(t, op, i) \vee active_n(t, i))))$

The meaning of all items described in the previous definition is fairly intuitive but not the last one. In concrete, Definition 1.5 implies that a transaction can be explicitly aborted by: the DBS_n; an explicit abort; or, a $\mathbf{crash}_n$ event. The failure of a DBS_n is modeled, from the transaction point of view, as if all currently executing transactions are set to $aborted$. However, when a crash occurs, all previous committed and aborted transactions, as they are already finished, remain unaltered.

The Figure 2 shows the possible state transitions concerning a transaction $t \in T$. In the figure, dashed lines represent state transitions which rely on the behavior of other transactions and the underlying concurrency control; and, solid lines represent transitions exclusively depending on the transaction code or the RP. If we focus on the continuous transition between $active$ and $aborted$, it is due to an $\mathbf{abort}_n(t)$. This action, as we are not modeling application explicit aborts will only be invoked by the RP.

As we have indicated at the beginning of the section, we consider that the database has a multiversion concurrency control algorithm providing SI. Under this level of isolation, reading from a snapshot means that a transaction t sees all the changes made by transactions that committed before t started with its first operation; in our case the $\mathbf{begin}_n(t)$ event. The results of its updates are installed when the transaction commits. However, a transaction t successfully commits iff there is not a concurrent transaction t' that has already committed and some of the written items by t' are also written by t. The previous rule can be reached following two equivalent –though different in the way updates are treated– the *first-updater-wins* or *first-committer-wins* rule [7]. Most commercial databases, such as PostgreSQL [42] or Oracle [34], implement the former and, hence, we provide a specification for the former one. To show the SI definition, we need the definition of the *writeset* of a transaction and also the definition of what a *snapshot* means.

Definition 2 (Writeset)**.** *Let* β_n *be a trace of* $Traces(DBS_n)$*. For every* $\beta_n[j]$*,* $0 \leq j \leq |\beta_n|$*, the writeset of a transaction* $t \in T$ *at* j *is defined as follows:* $WS_n(t, j) = \{op \in OP \mid \exists\, i, i \leq j \colon \nu_i = \mathbf{submit}_n(t, op) \wedge type(op) = \texttt{write}\}$.

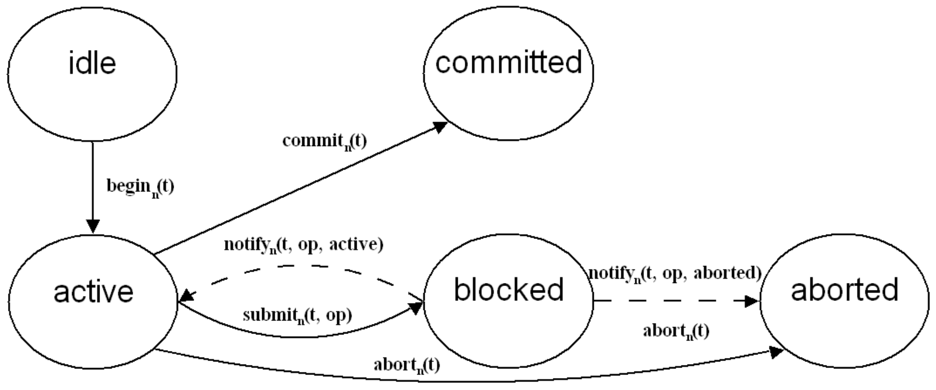

Figure 2: State transitions for a transaction $t \in T$.

Two writesets intersect if both contain at least one write operation over the same database item: $WS_n(t,j) \cap WS_n(t',k) \neq \emptyset \equiv \exists\, op \in WS_n(t,j), op' \in WS_n(t',k)\colon items(op) \cap items(op') \neq \emptyset$. We abuse the notation for indicating that a data item $X \in items(DB_n)$ belongs to a writeset $WS_n(t,j)$, $X \in WS_n(t,j)$, iff $\exists\, op, op \in WS_n(t,j)\colon X \in items(op)$.

In the next Definition, the concept of *snapshot* is introduced. The *snapshot* of the DB_n at some point of a trace comprises the latest versions of the data items until that point. These latest versions for each data item correspond to the last commit operation on each data item till that point of the trace.

Definition 3 (Snapshot)**.** *Let* β_n *be a trace of* $Traces(DBS_n)$*. For each* $\beta_n[j]$*,* $0 \leq j \leq |\beta_n|$*, the snapshot of* DB_n *at* j *is defined as follows:*

$Snapshot(\beta_n[j]) \equiv \cup_{X \in items(DB_n)}\, latestVer(X, \beta_n[j])$ *where*

$latestVer(X,\beta_n[j]) = \{X_t \in DB_n \mid$
$\exists\, i, i \leq j\colon (\nu_i = \mathbf{commit}_n(t) \wedge X \in WS_n(t,i) \wedge \forall\, k, i \leq k \leq j : \nu_k \notin \{\mathbf{commit}_n(t') \mid t' \in T, X \in WS_n(t',k)\})\}$

The following assumption provides conditions that a trace has to verify in order to be SI:

Assumption 2 (Snapshot Isolation - Safety)**.** *Let* $\nu_i = \mathbf{begin}_n(t)$*:*

1. $\nu_j = \mathbf{notify}_n(t, op, active, data) \wedge type(op) = \mathtt{read} \Rightarrow data \subseteq Snapshot(\beta_n[i])$[3]
2. $\nu_j = \mathbf{commit}_n(t) \Rightarrow \forall\, k, i \leq k \leq j\colon \nu_k \notin \{\mathbf{commit}_n(t') \mid t' \in T, WS_n(t,j) \cap WS_n(t',k) \neq \emptyset\}$

Assumption (2.1) indicates that a transaction reads the versions installed by committed transactions by the time of the $\mathbf{begin}_n(t)$ event; and, (2.2) shows that two concurrent transactions are allowed to make the commit if they have not write-conflicts.

The latter Assumption (2.2) is obtained as a consequence of the fact that the database system provides SI with the *first-updater-wins* rule. This rule enforces that concurrent write-conflicting operations remain blocked and only one (i.e. the first that performed the update) is allowed to continue its execution.

Assumption 3 (First-Updater-Wins)**.** $\nu_i = \mathbf{begin}_n(t) \wedge active_n(t,j) \Rightarrow (\forall\, k, i \leq k \leq j\colon \nu_k \notin \{\mathbf{commit}_n(t') | t' \in T, WS_n(t,j) \cap WS_n(t',k) \neq \emptyset\}) \wedge (\forall\, t', t' \in T(\beta_n[j]) \wedge \neg(committed_n(t',j) \vee aborted_n(t',j)) \wedge WS_n(t,j) \cap WS_n(t',j) \neq \emptyset\colon blocked_n(t',j))$

The first part of the previous implication states that only those transactions t' that have no conflict with an active transaction t may perform a commit; while the second one, says that there can never exist another active transaction t' conflicting with the active one.

[3]To be precise the $data$ subset defined in the consequent exclusively contains data items read by the transaction and, thus, it will not contain read operations over its own updates. The condition is given in such a way for the sake of simplicity.

Until now previous assumptions are actually safety requirements for a trace. However, the specification requires to determine progress properties for traces. More precisely, we need to derive the next state transitions (see Figure 2): from $blocked$ to $aborted$; and, from $blocked$ to $active$. In the next assumption, we consider that there is not an explicit $\mathbf{abort}_n(t)$ event for a transaction t nor $\mathbf{crash}_n$. This fact simplifies the presentation though it does not diminish the correctness of the proposed results.

Assumption 4 (Snapshot Isolation - Liveness)**.** *Let* $\nu_i = \mathbf{begin}_n(t)$*:*

1. $blocked_n(t, op, j) \wedge \exists\, k, i < k \leq j \colon \nu_k \in \{\mathbf{commit}_n(t') \mid t' \in T, WS_n(t,j) \cap WS_n(t',k) \neq \emptyset\}$
   $\Rightarrow \exists\, z, z > j \colon \nu_z = \mathbf{notify}_n(t, op, aborted)$

2. $blocked_n(t, op, j) \wedge type(op) = \texttt{read} \Rightarrow \exists\, z, z > j \colon \nu_z = \mathbf{notify}_n(t, op, active)$

3. $blocked_n(t, op, j) \wedge type(op) = \texttt{write}$
   $\wedge\, \forall\, k, i < k \leq j \colon \nu_k \notin \{\mathbf{commit}_n(t') \mid t' \in T, WS_n(t,j) \cap WS_n(t',k) \neq \emptyset\}$
   $\wedge\, \forall\, t', t' \in T \wedge (active_n(t',j) \vee blocked_n(t',j)) \colon WS_n(t',j) \cap WS_n(t,j) = \emptyset$
   $\Rightarrow \exists\, z, z > j \colon (\nu_z = \mathbf{notify}_n(t, op, active) \vee$
   $\nu_z \in \{\mathbf{notify}_n(t', op', active), \mathbf{commit}_n(t') \mid t' \in T, op' \in OP, WS_n(t,j) \cap WS_n(t',z) \neq \emptyset\})$

Assumption (4.1) states that a transaction gets eventually aborted if some conflictive transaction commits when it is blocked; (4.2) says that a transaction gets eventually active when it submits a read operation; and, (4.3) considers that if a transaction is blocked by a write operation and there are not any concurrent transactions which are in conflict with the transaction, and this situation persists, then the transaction eventually gets active. Note that Assumption (4.2) states that read-only transactions never get blocked. So, for the sake of simplicity this kind of transactions are not considered on the sequel.

The previous rules do not prevent the deadlock occurrence. When a group of transactions are deadlocked, and this situation is detected by the DBS_n, one of them is aborted (which one is left to the decision of the DBS_n) in order to resolve the situation. When that situation happens, the DBS_n generates a $\mathbf{notify}_n(t, op, aborted)$ for one (or more than one) chosen transaction t.

Finally, we introduce the definition of the *log* of committed transactions in the DBS_n. Each element in the log is a tuple including the transaction identifier, its writeset until the commit; and a number indicating the order in the *log*. This number may be interpreted as the snapshot version because each time a transaction commits a new snapshot is installed in the DB_n.

Definition 4. *Let* β_n *be a trace of* $Traces(DBS_n)$*. For all* $\beta_n[j]$, $0 \leq j \leq |\beta_n|$, *the log of* DBS_n *at* j *is recursively defined as follows:*

- $log(\beta_n[0]) = empty$
- $log(\beta_n[j+1]) = log(\beta_n[j]) \cdot \langle t, WS(t,j), |log(\beta_n[j])| + 1\rangle$ *iff* $\beta_n[j+1] = \beta_n[j] \cdot \mathbf{commit}_n(t)$.
- $log(\beta_n[j+1]) = log(\beta_n[j])$ *in other case.*

4 Group Communication System Specification

In this section, we introduce the GCS specification. The RP requires for its correct behavior that all messages are delivered in the same order to all available replicas which is provided by the *atomic broadcast* communication primitive provide by a GCS. The uniform total order broadcast provides the following properties[13, 15, 21]: (i) delivery integrity; (ii) no duplication; (iii) uniform total order; (iv) validity; and, (v) uniform agreement. However, all these properties do not cover the RP necessities since we are assuming *crash* failures. In particular, the strongest specification (uniform agreement and uniform total order) of this communication primitive does not prevent the contamination phenomenon [15]. This case consists in a *faulty* replica that reaches an inconsistent state (e.g., because it has not been able to deliver a previously broadcast message) and then broadcasts a total-order message before crashing and, thus, contaminates the rest of correct replicas. Hence, it is needed that for any two replicas the set of delivered messages must be one prefix of the other, or viceversa. This property is known as *prefix order*[4] delivery [15]. All these interesting properties will be formalized later on in an assumption.

[4]An alternative definition for this phenomenon is *gap free*. It does not allow gaps in the delivery sequence. This alternative is more appropriate when a *crash-recovery* model is considered, since prefix order precludes the joining of new nodes [15].

In the following, M denotes the set of possible messages. We will consider each message $m \in M$ is different. The RP at site $n \in N$ makes use of two primitives which conform the set of possible events of this component: $\mathbf{broadcast}_n(m)$ and $\mathbf{deliver}_n(m)$. The former one is used by RP at site n to broadcast in total order a message m. The second one allows the RP instance at site n to receive in total order the message m previously broadcast by some replica. Thus, the set of events of the GCS is, $Events(GCS)$=

$\{\mathbf{broadcast}_n(m)|n \in N, m \in M\} \cup$
$\{\mathbf{deliver}_n(m)|n \in N, m \in M\} \cup$
$\{\mathbf{crash}_n|n \in N\}$

In the following, we provide the assumptions the set of traces, $Traces(GCS)$, verifies. We define the function $site\colon Events(GCS) \rightarrow N$ which returns the site at which an event occurs. Let γ be a trace of $Traces(GCS)$. For each site $n \in N$ we define $mess_n$ as the sequence of messages delivered at site $n \in N$.

Definition 5. *Let γ be a trace of $Traces(GCS)$. For every $\gamma[j]$, $0 \leq j \leq |\gamma|$, the set of delivered messages in site $n \in N$ by the GCS at j is recursively defined as follows:*

- $mess_n(\gamma[0]) = empty$
- $mess_n(\gamma[j+1]) = mess_n(\gamma[j]) \cdot \langle m \rangle$ *iff* $\gamma[j+1] = \gamma[j] \cdot \mathbf{deliver}_n(m)$
- $mess_n(\gamma[j+1]) = mess_n(\gamma[j])$ *in other case*

In the following assumption we establish and formalize all the properties ensured by the GCS component in order to satisfy the requirements of the RP.

Assumption 5 (Prefix Order Atomic Broadcast)**.** *The GCS component satisfies the following properties:*

1. *(Crash Failures)* $site(\nu_i) = n \Rightarrow \nexists\, k, k < i\colon \nu_k = \mathbf{crash}_n$
2. *(Message Uniqueness)* $\nu_i = \mathbf{broadcast}_n(m) \wedge \nu_j = \mathbf{broadcast}_{n'}(m) \Rightarrow i = j$
3. *(Delivery Integrity)* $\nu_i = \mathbf{deliver}_n(m) \Rightarrow \exists\, n' \in N\colon (\exists\, j, j < i\colon \nu_j = \mathbf{broadcast}_{n'}(m))$
4. *(No Duplication)* $\nu_i = \mathbf{deliver}_n(m) \wedge \nu_j = \mathbf{deliver}_n(m) \Rightarrow i = j$
5. *(Prefix Order) For all $\gamma[i]$, $0 \leq i \leq |\gamma|$, and for any two sites $n, n' \in N$, either $mess_n(\gamma[i]) \preceq mess_{n'}(\gamma[i])$, or $mess_{n'}(\gamma[i]) \prec mess_n(\gamma[i])$*
6. *(Validity)* $n \in Correct(N) \wedge \nu_i = \mathbf{broadcast}_n(m) \Rightarrow \forall\, n', n' \in Correct(N)\colon (\exists\, j, j > i\colon \nu_j = \mathbf{deliver}_{n'}(m))$
7. *(Uniform Agreement)* $\nu_i = \mathbf{deliver}_n(m) \Rightarrow \forall\, n', n' \in Correct(N)\colon (\exists\, j\colon \nu_j = \mathbf{deliver}_{n'}(m))$

In the previous assumption condition (5.1) states that after a $\mathbf{crash}_n$ event the site $n \in N$ stops its activity; (5.2) indicates that messages are different; (5.3) and (5.4) state that every site delivers a message at most once and only if it was previously sent by some site; (5.5) guarantees that messages are delivered in the same total order without gaps even for faulty processes that always are a prefix of a correct site; (5.6) indicates that if a correct site invokes a broadcast event then all correct sites will eventually deliver the message; and, (5.7) states that if a site (correct or faulty) delivers a message then all correct sites will eventually deliver it.

5 Replication Protocol Description

In this Section the RP is described using a state transition system as introduced in Section 2.1. The RP at each site $n \in N$ uses for its execution the components introduced in the previous sections. Thus, for message exchange employs a GCS and for programming transactions uses its associated database system DBS_n.

As a rough outline of our RP proposal behavior, let us say that it is an eager update everywhere one [20]. A transaction is firstly executed at its delegate replica, which is determined by $t.site = n$ with $n \in N$. There is no message exchange and

Types:

$INFO = \mathbf{struct}\{start \in \mathbb{Z}^+, WS \in 2^{OP}, end \in \mathbb{Z}^+\}$

$M = T \times INFO$ // Message type

$SITE\_STATE = \{\text{alive}, \text{crashed}\}$

$STATUS = \{\text{idle}, \text{active}, \text{blocked}, \text{aborted}, \text{committed}\}$

Variables:

$\forall n \in N\colon channel_n = queue\_of(M)$, initially $channel_n =$ empty.

$\forall n \in N, \forall t \in T\colon info_n(t) \in INFO$, initially $info_n(t).start = 0$, $info_n(t).end = 0, info_n(t).WS = \emptyset$.

$\forall n \in N, \forall t \in T\colon sent_n(t) \in \{\text{true}, \text{false}\}$, initially $sent_n(t) =$ false.

$\forall n \in N\colon SEQ_n = queue\_of(M)$, initially $SEQ_n =$ empty.

$\forall n \in N\colon site\_state_n \in SITE\_STATE$, initially $site\_state_n =$ alive.

$\forall n \in N, \forall t \in T\colon status_n(t) \in STATUS$, initially $status_n(t) =$ idle.

$\forall n \in N\colon Ver_n \in \mathbb{Z}^+$, initially $Ver_n = 0$.

$\forall n \in N\colon ws\_run_n \in \{\text{true}, \text{false}\}$, initially $ws\_run_n =$ false.

Events =

$\{\mathbf{crash}_n \mid n \in N\} \cup \{\mathbf{deliver}_n(m) \mid n \in N, m \in M\} \cup \{\mathbf{discard\_ws}_n(t) \mid n \in N, t \in T\} \cup$
$\{\mathbf{end\_commit}_n(t) \mid n \in N, t \in T\} \cup \{\mathbf{execute\_op}_n(t, op) \mid n \in N, t \in T, op \in OP, t.site = n\} \cup$
$\{\mathbf{execute\_ws}_n(t) \mid n \in N, t \in T, t.site \neq n\} \cup$
$\{\mathbf{notify}_n(t, op, result, data) \mid n \in N, t \in T, op \in OP, result \in \{\text{active}, \text{aborted}\}, data \subseteq DB_n\} \cup$
$\{\mathbf{request\_commit}_n(t) \mid n \in N, t \in T, t.site = n\}$.

Transitions:

```
$\mathbf{execute\_op}_n(t, op)$
enabled ≡ $status_n(t) \in \{\text{idle}, \text{active}\} \wedge site\_state_n = \text{alive}$
              $\wedge \neg sent_n(t) \wedge \neg(ws\_run_n \wedge type(op) = \text{write})$.
action  ≡ if $status_n(t) = \text{idle}$ then
              $info_n(t).start \leftarrow Ver_n$
              $\mathbf{begin}_n(t)$
          if $type(op) = \text{write}$ then
              $info_n(t).WS \leftarrow info_n(t).WS \cup \{op\}$
          $\mathbf{submit}_n(t, op)$
          $status_n(t) \leftarrow \text{blocked}$.

$\mathbf{notify}_n(t, op, result, data)$
action  ≡ $status_n(t) \leftarrow result$.

$\mathbf{request\_commit}_n(t)$
enabled ≡ $status_n(t) = \text{active} \wedge \neg sent_n(t)$
          $\wedge site\_state_n = \text{alive}$.
action  ≡ $sent_n(t) \leftarrow true$
          $\mathbf{broadcast}_n(\langle t, info_n(t)\rangle)$.

$\mathbf{crash}_n$
action  ≡ $site\_state_n \leftarrow \text{crashed}$
          $\forall t \in T, status_n(t) \in \{\text{active}, \text{blocked}\}$:
              $status_n(t) \leftarrow \text{aborted}$
          $channel_n \leftarrow \text{empty}$
          $ws\_run_n \leftarrow \text{undef}$
          $\forall t \in T\colon info_n(t) \leftarrow \text{undef}$
          $\forall t \in T\colon sent_n(t) \leftarrow \text{undef}$.

$\mathbf{deliver}_n(m)$
action  ≡ $channel_n \leftarrow channel_n \cdot m$.

$\mathbf{execute\_ws}_n(t)$
enabled ≡ $status_n(t) = \text{idle} \wedge site\_state_n = \text{alive}$
              $\wedge \langle t, info(t)\rangle = \text{head}(channel_n)$
              $\wedge certification(\langle t, info(t)\rangle, SEQ_n)$.
action  ≡ $\forall t' \in getConflicts(\langle t, info(t)\rangle)$ :
              $\mathbf{abort}_n(t')$
              $status_n(t') \leftarrow \text{aborted}$
          $info_n(t) = info(t)$
          $\mathbf{begin}_n(t)$; $\mathbf{submit}_n(t, info_n(t).WS)$
          $status_n(t) \leftarrow \text{blocked}$
          $ws\_run_n \leftarrow \text{true}$.

$\mathbf{discard\_ws}_n(t)$
enabled ≡ $site\_state_n = \text{alive} \wedge \langle t, info(t)\rangle = \text{head}(channel_n)$
              $\wedge \neg certification(\langle t, info(t)\rangle, SEQ_n)$.
action  ≡ $channel_n \leftarrow \text{tail}(channel_n)$

$\mathbf{end\_commit}_n(t)$
enabled ≡ $status_n(t) = \text{active} \wedge site\_state_n = \text{alive}$
              $\wedge \langle t, info(t)\rangle = \text{head}(channel_n)$.
action  ≡ $channel_n \leftarrow \text{tail}(channel_n)$
          $Ver_n \leftarrow Ver_n + 1$
          $info_n(t).end \leftarrow Ver_n$
          $SEQ_n \leftarrow SEQ_n \cdot \langle t, info_n(t)\rangle$
          $\mathbf{commit}_n(t)$
          $status_n(t) \leftarrow \text{committed}$
          $ws\_run_n \leftarrow \text{false}$.
```

Auxiliary Functions:

$\mathbf{getConflicts}(\langle t, info(t)\rangle) =$
$\{t' \in T \mid status_n(t') \in \{\text{active}, \text{blocked}\} \wedge info_n(t').WS \cap info(t).WS \neq \emptyset\}$.

$\mathbf{certification}(\langle t, info(t)\rangle, SEQ) \equiv$
$\nexists \langle t', info(t')\rangle$ **in** $SEQ\colon info(t').end > info(t).start \wedge info(t').WS \cap info(t).WS \neq \emptyset$.

Figure 3: State transition system for the RP.

all operations are issued on the local DBS_n that constitutes a *local transaction*. When the RP requests the commit of the transaction, the interaction with the rest of sites is started. All the updates are grouped (writeset) and sent, using the atomic broadcast, to the rest of available replicas. Upon its delivery, it is needed to pass a test, called *certification*, just to check if the incoming writeset can be applied or not. The certification, roughly speaking, consists in detecting if there are any conflict between concurrent, though committed (and, hence, contained in the *log* of the DBS_n) writesets and the incoming one. If so, the message will be discarded (in the case of its delegate replica the transaction will be rolled back) and, otherwise, a *remote transaction* is started to apply, and commit, the writeset of transaction t (in the case of the delegate site it will be committed). Thanks to the total-order delivery, the outcome of the certification will be the same for every delivered message. Thus, transactions will be committed in the same order at all sites and the database logs grow in in the same order. In this section, we will explain in the following subsections the variables, initial values, events, and transitions that compose such state transition system in a more detailed manner.

5.1 Variables and Initial Values

The variables being used by the RP at each site $n \in N$ are:

- $status_n(t)$: This variable holds the current state of each one of the known transactions $t \in T$ in each replica. Valid states are: $idle$ (transaction not started), $active$ (active transaction), $blocked$ (the transaction is waiting for the completion of a given operation), $aborted$ (the transaction has been aborted), or $committed$ (the transaction has been committed). All transactions are initialized to the $idle$ status value.

- $info_n(t)$: Each replica keeps track, as a data structure, of some relevant attributes of a transaction $t \in T$. Such attributes are the $start$ and end logical timestamps (needed for certification purposes), and the writeset associated to such a transaction (attribute WS). Initially, the logical timestamps are set to zero, and the WS is an empty set.

- $sent_n(t)$: Boolean flag that holds a true value when the writeset of a transaction $t \in T$ has already been broadcast. It holds a false initial value for all transactions.

- $channel_n$: This variable models the incoming channel of delivered messages in each site. Such channel is represented by a queue of messages, since delivery order is important, as already described in Section 4. The usual $head()$ and $tail()$ functions for queues are used to handle this variable. Thus, $head()$ returns the first element of the queue, whilst $tail()$ returns all queue elements excepting the first one. Each message $m \in M$ in $channel_n$ contains the values $\langle t, info(t) \rangle$ where t is the transaction identifier and $info(t)$ is the value of $info_{t.site}(t)$ when the message was sent. Initially, this queue is assumed empty.

- Ver_n: The database version at each site. Such database versions are increased each time a transaction is committed. This variable plays the role of a logical timestamp. Whenever a transaction t starts at its delegate replica, $t.site = n$, Ver_n is stored in $info_n(t).start$. The initial value of this variable is zero at all sites.

- SEQ_n: This variable models a local queue of already committed transactions in each replica. Such queue is needed for certifying the incoming messages. Each element contained in SEQ_n is of the form $\langle t, info(t) \rangle$ that corresponds to a transaction t, and its respective $info_n(t)$ when it committed at site $n \in N$. Its main difference with the information received in the message is that the $info_n(t).end$ field stores the Ver_n value when the transaction committed. Initially, it is an empty queue in each site.

- ws_run_n: This is a boolean variable initially false that is set to true whilst any remote transaction is being applied at a given site. This variable gives priority to the application of remote transactions; thus, preventing new local write accesses from progressing. Initially, as there are no remote writesets, it is set to false.

- $site\_state_n$: It holds the current state of each replica site. Valid state values are $alive$ or $crashed$. All system sites are assumed initially $alive$.

We assume that after a $\mathbf{crash}_n$ event some variables keep their associated values, this facilitates the presentation of some properties. More precisely, the $status_n(t)$, Ver_n and SEQ_n are are assumed to be kept in stable storage. All other variables are lost in case of crash failure and get an $undef$ or $empty$ value in such case, as it can be seen in the action associated to the $\mathbf{crash}_n$ event.

5.2 Events

The events in RP are shown in the corresponding section of Figure 3. There are a group of events such as $\mathbf{execute\_op}_n(t, op)$ and $\mathbf{request\_commit}_n(t)$ events which are only allowed to be executed at the delegate replica of t, i.e. $t.site = n$. The $\mathbf{execute\_ws}_n(t)$ event will be executed at all sites n such that $t.site \neq n$. This fact enforces the different performance of local transactions versus remote ones.

5.3 Transitions

We are going to describe, in a more detailed manner than the already described at the beginning of this section, the different transitions presented in Figure 3.

A local transaction t, $t.site = n$, is started by the first invocation of the $\mathbf{execute\_op}_n(t, op)$, this changes $status_n(t)$ from $idle$ to $active$ and the associated $info_n(t).start$ is set to Ver_n. As it will be explained later, this action can not be invoked if $ws\_run_n = \texttt{true}$ and the requested operation is an update one or $sent_n = \texttt{true}$. Additionally, the $\mathbf{begin}_n(t)$ event of the underlying DBS_n component is used in order to start the transaction. Finally, the operation is submitted to the DBS_n component and the transaction status is set to $blocked$. Notice that the writeset of t is dynamically built provided that the operation submitted is a write. The transaction remains blocked until the underlying DBS_n component fires its $\mathbf{notify}_n(t, op, result)$ event which is paired to the same event in the RP, i.e. no enabling condition is needed for this event. When this happens, the transaction $status$ is set to the $result$ parameter of such event. This implies that the transaction gets again $active$ or that it has been $aborted$. Note that the $\mathbf{execute\_op}_n(t, op)$-$\mathbf{notify}_n(t, op, result)$ pair of events can be repeated many times for the transaction.

The $\mathbf{request\_commit}_n(t)$ event will be called, once all operations of the local transaction have finished. To enable such event, such replica should be $alive$, the transaction must be $active$ (i.e., all its previous operations have received their notification event), and the $sent_n(t)$ flag should be false. This event sends the transaction identifier t with its associated data structure $info_n(t)$ to the rest of available replicas using the atomic broadcast. Additionally, the variable $sent_n(t)$ is set to true and, thus, prevents to perform new operations in the DBS_n or sending several messages for the same transaction t. Thus, the writeset sent in $\langle t, info(t)\rangle$ message will be the same for transaction t and this message will be unique at every site.

The message will be eventually delivered by the GCS component to all available replicas n', using its $\mathbf{deliver}_{n'}(m)$ event that matches the same event of the RP. The delivered message is appended to the $channel_{n'}$ local queue variable. Once the message is delivered and no prior message exists in the $channel_{n'}$ variable, it can follow different execution paths in the RP depending on whether the transaction is local or remote. Let us start with the latter, the writeset contained in $info(t).WS$ must be certified. The $certification(\langle t, info(t)\rangle, SEQ_{n'})$ function (which is formally shown as a logical predicate in Figure 3) is in charge of this, returning true if the given writeset does not intersect with any of the writesets of transactions, say $t' \in SEQ_{n'}$, whose $t'.end$ value is greater than $info(t).start$; otherwise, returning false. If the certification fails, the message will be discarded by way of $\mathbf{discard\_ws}_{n'}(t)$. In other case, the included writeset in $info(t)$ has to be applied, by way of $\mathbf{execute\_ws}_{n'}(t)$, as a remote transaction. It is worth noting that there can be local transactions potentially conflicting with $info(t).WS$ that are detected by the $getConflicts(\langle t, info(t)\rangle)$ function (see its formal definition in Figure 3). Each local transaction returned in $getConflicts(\langle t, info(t)\rangle)$ is aborted in order to avoid the potential blocking of the execution of the remote transaction by a conflict with a local transaction. Once a writeset is certified, its associated remote transaction must be eventually committed. In addition, the value of $ws\_run_{n'}$ is set to true avoiding that non-aborted local transactions perform update operations and, again, avoiding the existence of new conflicts with the remote transaction. Once the writeset has been applied, its respective $\mathbf{notify}_{n'}(t, op, result)$ event will be executed, and, it is important to emphasize its associated $result$ will be $active$. Hence, the $\mathbf{end\_commit}_{n'}(t)$ will be eventually called to commit the transaction in the underlying $DBS_{n'}$ component, increase the version ($Ver_{n'}$), and allowing the execution of update operations or the application of new remote transactions in $DBS_{n'}$. In the case of the delivery of the message to its delegate replica, i.e. $n = t.site$, it can follow two different paths as well. If the associated message reaches the first position of $channel_n$ and transaction t is still $active$, i.e., no other previous delivered transaction has a conflict with t, the transaction will be directly committed (it does not need a certification) by way of the $\mathbf{end\_commit}_n(t)$ event with exactly the same effects as in the case of a remote transaction. Otherwise, the certification of another previously concurrent delivered remote transaction has a conflict with t that resulted in the abortion of t. Thus, message will be discarded by the invocation of the $\mathbf{discard\_ws}_n(t)$.

The variable SEQ_n has to be built so that all replicas reach the same decision on the $certification(\langle t, info(t)\rangle, SEQ_n)$ function. The SEQ_n variable is only modified when a transaction is committed, i.e. in the $\mathbf{end\_commit}_n(t)$ event, and the respective $\langle t, info_n(t)\rangle$ is appended at the end. Transactions are applied and committed in the order they were certified. This order is completely determined by the total-order delivery provided by the GCS component. Therefore, it is easy to show that the way elements are appended is the same thanks to the way the RP execute transactions (this will be formally shown in Section 7).

Regarding to fault-tolerance issues, we should consider the $\mathbf{crash}_n$ event that matches the one provided in the GCS and DBS_n interfaces, and that is generated by the environment and applied in all the system components at once. When such event is generated, the corresponding $site\_state_n$ variable is set to $crashed$ and this means that all RP events become disabled. This reflects the case when a given a replica becomes unavailable and completely stops its activity. To complete the description of the state transition system of RP we assume *weak-fairness* [31] for the events: $\mathbf{execute\_ws}_n(t)$, $\mathbf{discard\_ws}_n(t)$ and $\mathbf{end\_commit}_n(t)$.

6 Correctness Criteria

In this Section, we introduce the safety and liveness criteria the whole system (RP, GCS and for all $n \in N$ their respective DBS_n), following the interaction and composition rules depicted before, must satisfy to ensure its correct behavior. The safety criterion, establishes that for any pair of sites the *log* of committed transactions in their respective database systems is either the prefix of the other or viceversa. As database systems provide SI, this criterion implies that each database replica at every site has installed (or generated) the same snapshots (as it has been seen in Definition 3) in the very same order. Therefore, each database reaches the same state, at commit time, for every executed transaction. The liveness criterion must ensure that if a site commits a transaction, it will be eventually committed at every correct replica. Under this criterion, all correct and available databases do not lose any committed update in any database of the system.

Let us formalize the previous criteria according to the components present in the replicated system: the RP; the GCS; and, for all $n \in N$, the DBS_n. Let α be a fair execution of the RP and s_z a reachable state of α. Now, we assume that for all $n \in N$, $\beta_n(\alpha) \in Traces(DBS_n)$, i.e. they verify every assumption given in Section 3.

Safety Criterion (Database Prefix Order Consistency)**.** *For every fair execution α of the RP, every reachable state s_z of α and for any pair of sites $n, n' \in N$: either $log(\beta_n(\alpha[z])) \preceq log(\beta_{n'}(\alpha[z]))$ or viceversa.*

Liveness Criterion (Uniform Commit)**.** *For every fair execution α of the RP and for all $n \in N$: $\nu_i = \mathbf{commit}_n(t)$ in $\beta_n(\alpha) \Rightarrow \forall\, n', n' \in Correct(N)$: $\mathbf{commit}_{n'}(t)$ in $\beta_{n'}(\alpha)$.*

The safety criterion is also very important to determine the final isolation level achieved by the RP for committed transactions. We have stated at the beginning of this work that RP provides GSI [17]. Actually, GSI is an extension of SI best suited for replicated environments. The GSI level allows the use of *older* snapshots of the database, facilitating its replicated implementation. A transaction may receive a snapshot that happened in the system before the time of its first operation (instead of its current snapshot as in SI). To commit a transaction it is necessary, as in SI, that no other update operation of recently committed transactions conflicts with its update operations. Thus, a transaction can observe an older snapshot of the database but the write operations of the transaction are still valid update operations for the database at commit time. Many of the desirable properties of SI remain also in GSI, in particular, read-only transactions never became blocked and neither they cause update transactions to block or abort.

Let us see how the previous concept of GSI level can be applied to the system. Suppose that a transaction t starts its execution at its delegate replica ($t.site = n$) where no update operations have occurred, i.e. its $Ver_n = 0$. Prior to this, a transaction t', whose delegate replica is $n' \neq n$, has been certified and committed at all available replicas but n, due to, e.g., a communication delay in the propagation of messages by the GCS to n. Let us assume that the associated $info(t').WS$ contains data item X and, hence, a new version of this item ($X_{t'}$) has been installed in the system. If transaction t reads data item X it will read the version X_0 instead of the already committed and installed in the system. Moreover, if a transaction t'' concurrently starts at n', though after t' has been committed, it will read version $X_{t'}$. It is easy to see that as transactions firstly performs their operations at their delegate replica, it is more likely to occur that they will get older snapshots than the ones already installed in the system. As RP follows the sequential commit of transactions in the order they are committed, and is a kind of ROWAA protocol [20], it satisfies the safety criterion for the executed transactions which is a sufficient condition for obtaining GSI level for the committed transactions. The proof of this fact is given in [19].

7 Correctness Proof

In the following, it is developed the correctness proof of RP. We have split the correctness proof into two parts "Safety Criterion" and "Liveness Criterion" as defined in the previous Section. In order to tackle with this we have defined a set of Properties and Lemma that will help in the proof of both criteria. All of them, have been proved and their proof is either shown in the body of this work, whenever it contributes to the clarity of the correctness criteria, or, otherwise, at the end, in Appendix A.

Let us start with the verification of the "Safety Criterion". Firstly, it is needed to check that the state of a transaction t in the DBS_n is properly maintained by the RP instance running at n. Hence, when the message containing the the updates of t is received, it is because t requested the commit at its delegate replica. Moreover, this update information remains exactly the same when t is committed. As these messages are delivered in the same order at all replicas, then we have that for every two replicas, say n and n', that delivered the update message, it must be shown that t will be committed (or discarded) in both. Moreover, as this process is done in the very same order the content of the SEQ_n $SEQ_{n'}$ will be the same at the moment the message of t is delivered. Of course, this does not imply a synchronous certification process between all replicas: transactions are processed at different speed on different replicas and the size of the sequencers varies. Nevertheless, it is needed to show with the aid of all of the mentioned features that the sequencer associated to a replica constitutes the prefix of another one or viceversa.

Regarding to the "Liveness Criterion", its first mission is to check that if a t is successfully certified then it is active and its associated update message is in the first position of the delivered messages queue. Of course, it must be ensured that this message will get eventually processed, i.e. preceding messages are certified and removed from the queue of delivered messages or the replica will crash. In the same way, it must ensure that if for any replica t is committed then it will eventually get committed at the rest of correct replicas which is the main reasoning of our "*Uniform Commit*" property. *To be completed*

7.1 Preliminaries

The RP uses the DBS_n component at each site $n \in N$ to execute transactions at that site. We consider that each DBS_n verifies the assumptions provided in Section 3 for its traces. However, it is necessary that the RP preserves the Assumption 1 in order to ensure a correct usage of that component. The first property indicates that a transaction is active in the RP at site $n \in N$ when the DBS_n component notifies such a fact. The second property states that the RP preserves the well-formed traces assumption for each DBS_n component.

Property 1. *Let* $\alpha = s_0\pi_1 s_1 \ldots s_{z-1}\pi_z s_z \ldots$ *be an arbitrary execution of the* RP*:*

$$s_z.status_n(t) = \textit{active} \Rightarrow \exists z_1 \leq z : \pi_{z_1} = \mathbf{notify}_n(t, op, active) \textit{ for some } op \in OP.$$

Property 2. *Let* α *be an arbitrary execution of the* RP *and* $\beta_n(\alpha)$*,* $n \in N$*, be its associated sequence of* $Events(DBS_n)$*.* $\beta_n(\alpha)$ *verifies the Assumption 1.*

If the transactions submitted by the RP to each DBS_n component are well-formed, then the DBS_n behaves accordingly with its specification and the rest of assumptions provided in Section 3 are preserved by the RP. In what follows, for each arbitrary execution α of the RP and $n \in N$: $\beta_n(\alpha) \in Traces(DBS_n)$.

The RP also uses the GCS for message communication among sites. The traces of the GCS verify the assumption given in Section 4. We also prove that the RP preserves those assumptions in order to guarantee the correct usage of this component. In this case, we only need to prove the Assumptions 5.1 (Crash Failures) and 5.2 (Message Uniqueness).

Property 3. *Let* α *be an arbitrary execution of the* RP *and* $\gamma(\alpha)$ *be its associated sequence of* $Events(GCS)$*.* $\gamma(\alpha)$ *verifies Assumptions 5.1 and 5.2.*

In what follows, for each arbitrary execution α of the RP: $\gamma(\alpha) \in Traces(GCS)$.

7.2 Proof of Safety Criterion

Some variables of the RP are used to keep track the underlying information about the transactions executed at DBS_n for a site $n \in N$. The next property indicates that the information RP maintains about the DBS_n is compatible with the definitions given in Section 3. In order to simplify the presentation, we make use of an auxiliary variable LOG_n in the next property. The content of this variable is exactly the same as the SEQ_n with the exception of the $info(t).start$ value of a committed transaction t.

Property 4. *Let* $\alpha = s_0\pi_1 s_1 \ldots s_{z-1}\pi_z s_z \ldots$ *be an arbitrary execution of the* RP. *Let* $\beta_n(\alpha)[j]$[5] $\in Traces(DBS_n)$, $n \in N$, *be its associated trace of* $Events(DBS_n)$ *until* s_z. *The next properties hold:*

1. $s_z.status_n(t) = idle \Rightarrow idle_n(t, j)$
2. $s_z.status_n(t) = active \Rightarrow active_n(t, j)$
3. $s_z.status_n(t) = blocked \Rightarrow blocked_n(t, op, j)$ *for some* $op \in OP$
4. $s_z.status_n(t) = committed \Rightarrow committed_n(t, j)$
5. $s_z.status_n(t) = aborted \Rightarrow aborted_n(t, j)$
6. $s_z.site\_state_n = alive \wedge t.site = n \wedge s_z.status_n(t) \neq idle \Rightarrow s_z.info_n(t).start = |log(\beta_n(\alpha)[i])| \wedge i \leq j \wedge \nu_i = begin_n(t)$
7. $s_z.site\_state_n = alive \wedge s_z.status_n(t) \neq idle \Rightarrow s_z.info_n(t).WS = WS_n(t, j)$
8. $s_z.site\_state_n = alive \wedge s_z.status_n(t) = committed \Rightarrow s_z.info_n(t).end = |log(\beta_n(\alpha)[i])| \wedge i \leq j \wedge \nu_i = commit_n(t)$
9. $s_z.Ver_n = |log(\beta_n(\alpha)[j])|$
10. $s_z.LOG_n = log(\beta_n(\alpha)[j])$

Property 4 indicates that the information regarding to current state of a transaction that is being executed at a DBS_n is properly kept by the RP instance executed at site $n \in N$. Besides, information about committed transactions, those appearing in the respective *log* is also kept in SEQ_n.

The next property states that if a $\langle t, info(t)\rangle$ message is received at any site in the system it is necessary that transaction t has requested its commit at its delegate replica ($t.site = n$). The different values contained in the message must coincide with their respective values when the transaction was executed. Every message has its associated $info(t).end = 0$ since this value will only be modified, in its respective variable, when the transaction is committed.

Property 5. *Let* $\alpha = s_0\pi_1 s_1 \ldots s_{z-1}\pi_z s_z \ldots$ *be an arbitrary execution of the* RP.

$\langle t, info(t)\rangle \in s_z.channel_{n'} \wedge t.site = n \Rightarrow \exists\, z_1 < z_2 < z_3 < z_4 \leq z : \pi_{z_1} = \mathbf{execute\_op}_n(t, \textit{first op of } t) \wedge \pi_{z_2} = \mathbf{notify}_n(t, \textit{last op of } t, active) \wedge \pi_{z_3} = \mathbf{request\_commit}_n(t) \wedge \pi_{z_4} = \mathbf{deliver}_{n'}(\langle t, info(t)\rangle) \wedge info(t).start = s_{z_1}.info_n(t).start \wedge info(t).WS = s_{z_2}.info_n(t).WS \wedge info(t).end = 0.$

Proof. As $\langle t, info(t)\rangle \in s_z.channel_{n'}$, then $\exists\, z_4 \leq z$: $\pi_{z_4} = \mathbf{deliver}_{n'}(\langle t, info(t)\rangle)$. By Assumption 5.4 (No Duplication) this event is the only one making such effect at s_{z_4}. By Assumption 5.3 (Delivery Integrity) there exists a previous $\mathbf{broadcast}_n(\langle t, info(t)\rangle)$ such that it is part of the action $\pi_{z_3} = \mathbf{request\_commit}_n(t)$ because this is the only one that broadcast that message being $t.site = n$. By Assumption 5.2 and Property 3 this action is the only one providing such an effect. In addition, $z_3 < z_4$. At s_{z_3-1}, $status_n(t) = active \wedge \neg sent_n(t)$. By Property 1: $\exists\, z_2 < z_3$: $\pi_{z_2} = \mathbf{notify}_n(t, \text{last op of } t, active)$. By definition, π_{z_2} is the last one for transaction t. As $s_{z_3}.sent_n(t) = \texttt{false}$, there is no possibility for $\mathbf{execute\_op}_n(t, op)$ after z_3. As those events are the only ones being able to modify variable $info_n(t).WS$ with $t.site = n$, then $info(t) = s_{z_3}.info_n(t).WS = s_{z_2}.info_n(t).WS$.
By Assumption 1.5, there exists a previous $\mathbf{submit}_n(t, \text{last op of } t)$ for which $\mathbf{notify}_n(t, \text{last op of } t, active)$ has been produced. Thus, by Assumption 1.4, there exists a unique $begin_n(t)$ which is part of the action of the event $\pi_{z_1} = \mathbf{execute\_op}_n(t, \text{first op of } t)$. Obviously, $z_1 < z_2$. By the effects of π_{z_1}, $s_{z_1}.info_n(t).start = s_{z_1}.Ver_n$. This event is the only one that modifies $info_n(t).start$, so $info(t).start = s_{z_1}.info_n(t).start$. All this reasoning is valid since there is no $\pi_{z'} = \mathbf{crash}_n$ in α with $z' \leq z_3$ nor $\pi_{z'} = \mathbf{crash}_{n'}$ with $z' \leq z$. □

[5] It is worth noting that $\beta_n(\alpha[z])$ is equivalent to $\beta_n(\alpha)[j]$.

Before an $\mathbf{end\_commit}_n(t)$ event the information concerning $info(t).start$ and $info(t).WS$ included in the message of the transaction t to be committed coincides with the current values of the $s_z.info_n(t).start$ and $s_z.info_n(t).WS$ variables respectively. The proof is trivial from the previous one taken into account that at site $t.site = n$ variable $sent_n(t) = \texttt{false}$ and after $\mathbf{request\_commit}_n(t)$ such variables have not been modified; and at site $t.site \neq n'$ the event $\mathbf{execute\_ws}_{n'}(t)$ copies $info(t)$ into $info_{n'}(t)$ variable and it will not be modified until the execution of $\mathbf{end\_commit}_{n'}(t)$. This is stated in the next property in a slightly weaker form.

Property 6. *Let $\alpha = s_0\pi_1 s_1 \ldots s_{z-1}\pi_z s_z \ldots$ be an arbitrary execution of the RP.*

$s_z.status_n(t) \neq idle \wedge s_z.site\_state_n = alive \wedge \langle t, info(t)\rangle \in s_z.channel_n \Rightarrow info(t).start = s_z.info_n(t).start \wedge info(t).WS = s_z.info_n(t).WS \wedge info(t).end = s_z.info_n(t).end$

The $\langle t, info(t)\rangle$ information included in SEQ_n contains the same values as their corresponding variables when the transaction t is committed.

Property 7. *Let $\alpha = s_0\pi_1 s_1 \ldots s_{z-1}\pi_z s_z \ldots$ be an arbitrary execution of the RP.*

$\langle t, info(t)\rangle \in s_z.SEQ_n \Rightarrow \exists z_1 \leq z : \pi_{z_1} = \mathbf{end\_commit}_n(t) \wedge info(t) = s_{z_1}.info_n(t)$

We can prove that if a transaction has been committed at two sites $n, n' \in N$, part of the information of the transaction stored at each SEQ_n and $SEQ_{n'}$ variables respectively is the same. In fact, using Property 5 the next property concludes that if $\langle t, info(t)\rangle \in s_z.SEQ_n$, then $info(t).WS$ is the writeset of the transaction executed at its delegated site $t.site$ by the time of request the commit, and $info(t).start$ is the version of the database $DBS_{t.site}$ when the transaction executed its first operation.

Property 8. *Let $\alpha = s_0\pi_1 s_1 \ldots s_{z-1}\pi_z s_z \ldots$ be an arbitrary execution of the RP.*

$\langle t, info(t)\rangle \in s_z.SEQ_n \wedge \langle t, info'(t)\rangle \in s_z.SEQ_{n'} \Rightarrow info(t).start = info'(t).start \wedge info(t).WS = info'(t).WS$

Proof. By Property 7, $\exists z_1, z_2 (\leq z)\colon \pi_{z_1} = \mathbf{end\_commit}_n(t) \wedge \pi_{z_2} = \mathbf{end\_commit}_{n'}(t) \wedge info(t) = s_{z_1}.info_n(t) \wedge info'(t) = s_{z_2}.info_n(t)$.
By Property 6, at s_{z_1-1} and s_{z_2-1}, which are the states at which π_{z_1} and π_{z_2} are enabled respectively, it is verified:
$\langle t, m(t)\rangle = head(s_{z_1-1}.channel_n) \wedge m(t).start = s_{z_1-1}.info_n(t).start \wedge m(t).WS = s_{z_1-1}.info_n(t).WS \wedge m(t).end = s_{z_1-1}.info_n(t).end$, and
$\langle t, m'(t)\rangle = head(s_{z_2-1}.channel_{n'}) \wedge m'(t).start = s_{z_2-1}.info_{n'}(t).start \wedge m'(t).WS = s_{z_2-1}.info_{n'}(t).WS \wedge m'(t).end = s_{z_2-1}.info_{n'}(t).end$.
By Property 5, as $\langle t, m(t)\rangle = head(s_{z_1-1}.channel_n)$ and $\langle t, m'(t)\rangle = head(s_{z_2-1}.channel_{n'})$ then $\langle t, m(t)\rangle = \langle t, m'(t)\rangle$.
Due to the fact that π_{z_1} and π_{z_2} do not modify the variables $info_n(t).start$, $info_{n'}(t).start$, $info_n(t).WS$ and $info_{n'}(t).WS$, then
$info(t).start = s_{z_1}.info_n(t).start = s_{z_1-1}.info_n(t).start = m(t).start = m'(t).start = s_{z_2-1}.info_{n'}(t).start = s_{z_2}.info_{n'}(t).start = info'(t).start$, and
$info(t).WS = s_{z_1}.info_n(t).WS = s_{z_1-1}.info_n(t).WS = m(t).WS = m'(t).WS = s_{z_2-1}.info_{n'}(t).WS = s_{z_2}.info_{n'}(t).WS = info'(t).WS$ □

If we move what it has been stated in Property 8 to the updates performed by a transaction in the database, it is equivalent to say that if they appear at two different sites they will be exactly the same. To end the proof of the safety criterion, it is necessary to prove that transactions have been committed in the same order. This fact can be shown thanks to the GCS properties that ensure the delivery of messages in the very same order to all sites and the RP since it processes messages in the same order they have been delivered.

In the next, we introduce a set of auxiliary variables in order that will help us during the correctness proof. Let $delivered_n$ be an auxiliary variable containing the delivered messages at site $n \in N$ by the event $\mathbf{deliver}_n(m)$; that is, $delivered_n \leftarrow delivered_n \cdot m$. Let $received_n$ be an auxiliary variable containing the messages the algorithm's events have handled at site $n \in N$, in particular $\mathbf{end\_commit}_n(t)$ and $\mathbf{discard\_ws}_n(t)$ events. When m is removed from $head(channel_n)$, it is included in $received_n$, i.e. $received_n \leftarrow received_n \cdot m$.

These new auxiliary variables have the initial values $s_0.delivered_n = s_0.received_n = empty$. Taken into account the GCS properties, the RP guarantees that the set of messages processed by it along with the set of pending messages constitutes the set of messages delivered by the GCS:

Property 9. *Let* $\alpha = s_0\pi_1 s_1 \ldots s_{z-1}\pi_z s_z \ldots$ *be an arbitrary execution of the* RP*. Let* $\gamma(\alpha[z]) \in Traces(GCS)$*, be its associated trace of* $Events(GCS)$ *until* s_z*. For each* $n \in N$ *the next properties hold:*

1. $s_z.delivered_n = mess_n(\gamma(\alpha[z]))$
2. $s_z.received_n \cdot s_z.channel_n \preceq s_z.delivered_n$

Thanks to the last property and the total-order delivery of messages, the set of processed messages by a replica is a prefix of the processed messages of another one or viceversa.

Property 10. *Let* $\alpha = s_0\pi_1 s_1 \ldots s_{z-1}\pi_z s_z \ldots$ *be an arbitrary execution of the* RP*. For all pairs of sites* $n, n' \in N$*, either* $s_z.received_n \preceq s_z.received_{n'}$ *or viceversa.*

Proof. By Property 9, $s_z.received_n \cdot s_z.channel_n \preceq mess_n(\gamma(\alpha[z]))$ and $s_z.received_{n'} \cdot s_z.channel_{n'} \preceq mess_{n'}(\gamma(\alpha[z]))$. By Assumption 5.5 (Prefix Order), $mess_n(\gamma(\alpha[z])) \preceq mess_{n'}(\gamma(\alpha[z]))$ or viceversa.
Therefore, $s_z.received_n \preceq s_z.received_{n'}$ or viceversa. □

The next property is the most important one introduced so far, it states that a transaction is certified (or discarded, respectively) in the same order at all available sites. Note that the certification process is asynchronous. Hence, one replica can be faster (i.e. process more messages) and certify more transactions. Nevertheless, the decision (commit or abort) for each certified transaction at every replica will be the same.

Property 11. *Let* $\alpha = s_0\pi_1 s_1 \ldots s_{z-1}\pi_z s_z \ldots$ *be an arbitrary execution of the* RP*. For all pairs of sites* $n, n' \in N$*:*
$s_z.received_n \preceq s_z.received_{n'} \Rightarrow \exists z_1 \leq z : s_z.SEQ_n = s_{z_1}.SEQ_{n'} \wedge s_z.received_n = s_{z_1}.received_{n'}$.

Proof. By induction over the length of α.

- *Basis.* At $\alpha = s_0$; for all $n \in N$: $s_0.received_n = empty$, and $s_0.SEQ_n = empty$. The property holds.
- *Hypothesis.* Assume the property holds at s_z.
- *Induction Step.* Let $(s_z, \pi_{z+1}, s_{z+1})$ be a transition of the RP. We study all the events π_{z+1} affecting the variables of the property in the following cases:
  Case 1: $s_z.received_n = s_z.received_{n'}$ and π_{z+1} at site n'.
  Case 2: $s_z.received_n \prec s_z.received_{n'}$ and π_{z+1} at site n.
  Case 3: $s_z.received_n = s_z.received_{n'}$ and π_{z+1} at site n.
  Case 4: $s_z.received_n \succ s_z.received_{n'}$ and π_{z+1} at site n'.
  It is sufficient to prove cases 1 and 2 because cases 3 and 4 are symmetric by interchanging n by n' in the Property 11.
  - Case 1: $\pi_{z+1} \in \{\mathbf{end\_commit}_{n'}(t), \mathbf{discard\_ws}_{n'}(t)\}$. By its effects, $s_z.received_n = s_{z+1}.received_n$ and $s_{z+1}.received_{n'} = s_z.received_{n'} \cdot \langle t, info(t)\rangle$. Thus $s_{z+1}.received_n \prec s_{z+1}.received_{n'}$. By induction Hypothesis and the fact $s_z.SEQ_n = s_{z+1}.SEQ_n$ does not change by the execution of π_{z+1}, the same s_{z_1} considered in the induction Hypothesis makes the property true at s_{z+1}.
  - Case 2: $\pi_{z+1} \in \{\mathbf{end\_commit}_n(t), \mathbf{discard\_ws}_n(t)\}$.
    * $\pi_{z+1} = \mathbf{end\_commit}_n(t)$. By the effects of its action:
      $s_{z+1}.received_n = s_z.received_n \cdot \langle t, info(t)\rangle$ and $s_{z+1}.SEQ_n = s_z.SEQ_n \cdot \langle t, s_{z+1}.info(t)\rangle$. Recall that $certification(\langle t, info(t)\rangle, s_z.SEQ_n) = \texttt{true}$.
      By induction Hypothesis: $\exists z_1 \leq z : s_z.SEQ_n = s_{z_1}.SEQ_{n'} \wedge s_z.received_n = s_{z_1}.received_{n'}$.
      Therefore, as $s_z.received_n \prec s_z.received_{n'}$, then $s_{z+1}.received_n \preceq s_{z+1}.received_{n'}$.
      Notice: $s_{z_1}.received_{n'} \cdot \langle t, info(t)\rangle = s_z.received_n \cdot \langle t, info(t)\rangle \preceq s_{z+1}.received_{n'}$.
      Let π_{z_2} be the event at n' such that $s_{z_2}.received_{n'} = s_{z_1}.received_{n'} \cdot \langle t, info(t)\rangle$. No other event modifying $received_{n'}$ has been executed between z_1 and z_2. Actually, what we do know, by inspection of the algorithm's code, is that the following does not change: $s_{z_2-1}.channel_{n'} = s_{z_1}.channel_{n'} \wedge s_{z_2-1}.SEQ_{n'} = s_{z_1}.SEQ_{n'}$.
      Thus, $s_{z_2-1} = s_{z_1}$, in particular $\langle t, info(t)\rangle = head(s_{z_2-1}.channel_{n'})$. The possible events taken $\langle t, info(t)\rangle$ into $s_{z_2}.received_{n'}$ are $\pi_{z_2} \in \{\mathbf{end\_commit}_{n'}(t), \mathbf{discard\_ws}_{n'}(t)\}$. Only $\pi_{z_2} = \mathbf{end\_commit}_{n'}(t)$ is possible because $certification(\langle t, info(t)\rangle, s_{z_2-1}.SEQ_n) = certification(\langle t, info(t)\rangle, s_z.SEQ_n)$. Therefore, by the effects of the event's action π_{z_2}:

$s_{z_2}.SEQ_{n'} = s_{z_2-1}.SEQ_{n'} \cdot \langle t, s_{z_2}.info(t)\rangle$; $s_{z_2}.Ver_{n'} = s_{z_2-1}.Ver_{n'}+1$; and, $s_{z_2}.info_{n'}(t).end = s_{z_2}.Ver_{n'}$.
By Property 8: $s_{z+1}.info_n(t).start = s_{z_2}.info_{n'}(t).start$; $s_{z+1}.info_n(t).WS = s_{z_2}.info_{n'}(t).WS$.
As $|s_{z+1}.SEQ_n| = |s_{z_2}.SEQ_{n'}|$ (Recall $|SEQ_n| = Ver_n$) then $s_{z+1}.info_n(t).end = s_{z_2}.info_{n'}(t).end$.
In conclusion:
$s_{z+1}.received_n \preceq s_{z+1}.received_{n'}$
$\Rightarrow$
$\exists\, z_2 \leq z+1$:
$s_{z+1}.received_n = s_z.received_n \cdot \langle t, info(t)\rangle = s_{z_2}.received_{n'} \wedge$
$s_{z+1}.SEQ_n = s_z.SEQ_n \cdot \langle t, s_{z+1}.info(t)\rangle = s_{z_2}.SEQ_{n'}$.

$*$ $\pi_{z+1} = \mathbf{discard\_ws}_n(t)$. By the effects of its action:
$s_{z+1}.received_n = s_z.received_n \cdot \langle t, info(t)\rangle$. Notice that $certification(\langle t, info(t)\rangle, s_z.SEQ_n) = \texttt{false}$.
By induction Hypothesis: $\exists\, z_1 \leq z$: $s_z.SEQ_n = s_{z_1}.SEQ_{n'} \wedge s_z.received_n = s_{z_1}.received_{n'}$.
Therefore, as $s_z.received_n \prec s_z.received_{n'}$, then $s_{z+1}.received_n \preceq s_{z+1}.received_{n'}$.
Notice: $s_{z_1}.received_{n'} \cdot \langle t, info(t)\rangle = s_z.received_n \cdot \langle t, info(t)\rangle \preceq s_{z+1}.received_{n'}$.
Let π_{z_2} be the event at n' such that $s_{z_2}.received_{n'} = s_{z_1}.received_{n'} \cdot \langle t, info(t)\rangle$. No other event modifying $received_{n'}$ has been executed between z_1 and z_2. Following the same reasoning as before, we obtain that $s_{z_2-1} = s_{z_1}$, in particular $\langle t, info(t)\rangle = head(s_{z_2-1}.channel_{n'})$. The possible events taken $\langle t, info(t)\rangle$ into $s_{z_2}.received_{n'}$ are $\pi_{z_2} \in \{\mathbf{end\_commit}_{n'}(t), \mathbf{discard\_ws}_{n'}(t)\}$. Only $\pi_{z_2} = \mathbf{discard\_ws}_{n'}(t)$ is possible because $certification(\langle t, info(t)\rangle, s_{z_2-1}.SEQ_n) = certification(\langle t, info(t)\rangle, s_z.SEQ_n)$. Therefore, by the effects of the event's action π_{z_2}:
$s_{z_2}.SEQ_{n'} = s_{z_2-1}.SEQ_{n'}$. In conclusion:
$s_{z+1}.received_n \preceq s_{z+1}.received_{n'}$
$\Rightarrow$
$\exists\, z_2 \leq z+1$:
$s_{z+1}.received_n = s_z.received_n \cdot \langle t, info(t)\rangle = s_{z_2}.received_{n'} \wedge$
$s_{z+1}.SEQ_n = s_z.SEQ_n = s_{z_2}.SEQ_{n'}$.

$\square$

To conclude this part of the correctness proof, it is needed to show that the way successfully certified transactions are stored in SEQ_n and constitute, thanks to the Property 11, with any other replica n' a prefix ($SEQ_n \preceq SEQ_{n'}$) or viceversa is reflected in the way these transactions are committed in the database. In other words, it is needed to show that the database logs of committed transactions associated with any pair of replicas DBS_n and $DBS_{n'}$ either one is the prefix of the other ($LOG_n \preceq LOG_{n'}$) or viceversa.

Theorem 1 (Safety - Database Prefix Order Consistency)**.** *The RP verifies the safety criterion: "Database Prefix Order Consistency".*

Proof. Let $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ be an arbitrary execution of the RP. By Property 11, for all pair of sites $n, n' \in N$: $s_z.received_n \preceq s_z.received_{n'} \Rightarrow s_z.SEQ_n \preceq s_z.SEQ_{n'}$. By Property 10, either $s_z.received_n \preceq s_z.received_{n'}$ or viceversa.

Let $\beta_n(\alpha) \in Traces(DBS_n)$ and $\beta_{n'}(\alpha) \in Traces(DBS_{n'})$ be the associated traces of $Events(DBS_n)$ and $Events(DBS_{n'})$ respectively. By LOG_n definition, either $s_z.LOG_n \preceq s_z.LOG_{n'}$ or viceversa and by Property 4.10 (recall that $s_z.LOG_n = log(\beta_n(\alpha[z]))$): $log(\beta_n(\alpha[z])) \preceq log(\beta_{n'}(\alpha[z]))$ or viceversa. $\square$

7.3 Proof of Liveness Criterion

Upon the total-order delivery of a message, it is needed to prove that the writeset contained in it is processed in the same order at all available replicas. Hence, we need to study carefully the role of the certification decision. By the specification of RP, it is proved that the precondition of an $end\_commit_n(t)$ event implies that the evaluation of the $certification(t, SEQ_n)$ function is true. On the other hand, it is worth noting that at the delegate site of a transaction, the protocol does not check the certification function if the transaction is active before its commit (i.e. no other previous successfully certified transaction has aborted it). The next property states that given a transaction t if its associated $status_n(t)$ is equal to active and its associated message is in the first position of the sequence of messages to be processed (i.e. it is ready to commit) then it has been successfully certified.

Property 12. *Let $\alpha = s_0\pi_1 s_1 \ldots s_{z-1}\pi_z s_z \ldots$ be an arbitrary execution of the RP.*

$$s_z.status_n(t) = \textit{active} \wedge \langle t, info(t)\rangle = \textit{head}(s_z.channel_n) \Rightarrow certification(\langle t, info(t)\rangle, s_z.SEQ_n).$$

From the last property, it can be inferred that every committed transaction has been successfully certified. However, this is not enough to guarantee that every successfully certified transaction is eventually committed. The following property points out this fact: for every satisfyingly certified transaction the RP ensures that its status becomes active and, hence, is able to commit.

Property 13. *Let $\alpha = s_0\pi_1 s_1 \ldots s_{z-1}\pi_z s_z \ldots$ be an arbitrary execution of the RP.*

$$t.site = n \wedge \langle t, info(t)\rangle = \textit{head}(s_z.channel_n) \wedge certification(\langle t, info(t)\rangle, s_z.SEQ_n) \Rightarrow s_z.status_n(t) = \textit{active}.$$

Based on Properties 12 and 13, it is shown the execution flow of transactions at each site whenever a writeset is about to be applied. The RP must ensure that the transaction commits so it alters the execution flow of the rest of local transactions. This task is split into two parts: all conflicting transactions are immediately aborted; and, on the other hand, all local transactions attempting to update a data item are stopped (thus, avoiding new conflicting sources) until the writeset is applied.

Lemma 1. *Let α be a fair execution of the RP and let s_z be a reachable state in α.*

$certification(\langle t, info(t)\rangle, s_z.SEQ_n) \wedge \langle t, info(t)\rangle = head(s_z.channel_n) \Rightarrow \exists z' > z\colon s_{z'}.status_n(t) = committed \vee s_{z'}.site\_state_n = crashed$.

Proof. By contradiction. We assume the Lemma does not hold: $\forall z' \geq z\colon s_{z'}.status_n(t) \neq committed \wedge s_{z'}.site\_state_n = alive$. We consider the following cases:

- Case 1. $t.site = n$. Let s_z be the first state verifying the antecedent. By Property 13, $s_z.status_n(t) = active$. The event $\mathbf{end\_commit}_n(t)$ is enabled at s_z. No other action, except $\mathbf{crash}_n$ or $\mathbf{end\_commit}_n(t)$, can modify the variable $channel_n$ when $\langle t, info(t)\rangle = head(s_z.channel_n)$; or $status_n(t)$. Recall, $\forall z' \geq z\colon s_{z'}.site\_state_n = alive$, and $\mathbf{crash}_n$ event does not happen in α. The same happens with $\mathbf{end\_commit}_n(t)$ because $s_{z'}.status_n(t) \neq committed$. Thus, $\forall z' \geq z$, the event $\mathbf{end\_commit}_n(t)$ is enabled at $s_{z'}$. By weak-fairness of α, $\exists z' > z\colon \pi_{z'} = \mathbf{end\_commit}_n(t)$. Therefore, $s_{z'}.status_n(t) = committed$. The Lemma is verified in such a case.

- Case 2. $t.site \neq n$. Let s_z be the first state verifying the antecedent. Then, $s_z.status_n(t) = idle$. The event $\mathbf{execute\_ws}_n(t)$ is enabled at s_z. No other action, except $\mathbf{crash}_n$ or $\mathbf{end\_commit}_n(t)$, can modify the variable $channel_n$ when $\langle t, info(t)\rangle = head(s_z.channel_n)$, SEQ_n, ws_run_n, or $status_n(t)$. As $\forall z' \geq z\colon s_{z'}.site\_state_n = alive$, the $\mathbf{crash}_n$ event does not happen in α. The same happens with $\mathbf{end\_commit}_n(t)$ because $s_{z'}.status_n(t) \neq committed$. Thus, $\forall z' \geq z$, the event $\mathbf{execute\_ws}_n(t)$ is enabled at $s_{z'}$. By weak-fairness of α, $\exists z_1 > z\colon \pi_{z_1} = \mathbf{execute\_ws}_n(t)$. Let $\beta_n(\alpha)[j] \in Traces(DBS_n)$ be the associated trace to α at the reachable state s_{z_1-1} (the previous state before π_{z_1} execution). We consider now the effects of the execution of π_{z_1}:
  $\forall t' \in getConflicts(\langle t, info(t)\rangle)\colon abort_n(t')$, $s_{z_1}.status_n(t') = aborted$
  $s_{z_1}.info_n(t) = info(t)$
  $\mathbf{begin}_n(t)$, $\mathbf{submit}_n(t, s_{z_1}.info_n(t))$
  $s_{z_1}.status_n(t) = blocked$
  $s_{z_1}.ws\_run_n = \texttt{true}$.

  We consider $\{t'_1, ..., t'_k\} = getConflicts(\langle t, info(t)\rangle)$. Notice that $getConflicts(\langle t, info(t)\rangle)$ is evaluated at s_{z_1-1}. Thus, the associated trace $\beta_n(\alpha)$ at the reachable state s_{z_1} is ($j' = j + k + 2$):
  $\beta_n(\alpha)[j'] = \beta_n(\alpha)[j] \cdot \mathbf{abort}_n(t'_1) \cdot \cdots \cdot \mathbf{abort}_n(t'_k) \cdot \mathbf{begin}_n(t) \cdot \mathbf{submit}_n(t, info(t).WS)$.

  In the trace $\beta_n(\alpha)$ of the DBS_n, it is verified the following at j':
  (a) $blocked_n(t, info(t).WS, j') \wedge type(info(t).WS) = \texttt{write}$
  (b) $\wedge \forall k, j'-1 < k \leq j'\colon \nu_k \notin \{\mathbf{commit}_n(t') \mid t' \in T, WS_n(t, j) \cap WS_n(t', k) \neq \emptyset\}$

(c) $\wedge \forall\, t' \in T$ being $(active_n(t',j') \vee blocked_n(t',j'))\colon WS_n(t',j') \cap WS_n(t,j') = \emptyset$.

The condition (a) is trivial by Property 4 and Definition 2. The condition (b) is verified by construction of $\beta_n(\alpha)[j']$ because $\nu_{j'} = \mathbf{submit}_n(t, info(t).WS)$. We need further explanation in condition (c). Consider $\exists\, t' \in T$ being $(active_n(t',j') \vee blocked_n(t',j'))\colon WS_n(t',j') \cap WS_n(t,j') \neq \emptyset$. By construction of $\beta_n(\alpha)[j']$, t' verifies $(active_n(t',j) \vee blocked_n(t',j)) \wedge WS_n(t',j) \cap WS_n(t,j') \neq \emptyset$. By Property 4 and the fact that $WS_n(t,j') = info(t).WS$, the next holds:
$\exists\, t' \in T\colon s_{z_1-1}.status_n(t') \in \{active, blocked\} \wedge s_{z_1-1}.info_n(t').WS \cap info(t).WS \neq \emptyset$.
By definition of $getConflicts(\langle t, info(t)\rangle)$ at s_{z_1-1}, then $t' \in getConflicts(\langle t, info(t)\rangle)$ and $t' \in \{t'_1, \ldots, t'_k\}$. Therefore, $aborted_n(t',j')$. This is against the initial supposition of $active_n(t',j') \vee blocked_n(t',j')$.

The conditions (a), (b) and (c) are the antecedent of Assumption 4.3. Following the Assumption 4.3:
$\exists\, j'' > j'\colon \nu_{j''} = \mathbf{notify}_n(t, info(t).WS, active) \vee$
$\nu_{j''} \in \{\mathbf{notify}_n(t', op', active), \mathbf{commit}_n(t') \mid op' \in OP, t' \in T, WS_n(t,j') \cap WS_n(t',j'') \neq \emptyset\}$.

Recall that $\mathbf{end\_commit}_n(t)$ does not happen in α. Then, $\forall\, z' \geq z_1\colon s_{z'}.ws\_run = \texttt{true} \wedge \langle t, info(t)\rangle = head(s_{z'}.channel_n)$; then, $\pi_{z'} \neq end\_commit_n(t')$ for any $t' \in T$. This last consideration yields that $\nu_{j''} \neq \mathbf{commit}_n(t')$ for any $t' \in T$. If $\nu_{j''} = \mathbf{notify}_n(t', op', active)$ for some $t' \in T$, by Assumption 1.4, the previous event is $\mathbf{submit}_n(t', op')$ and $type(op') = \texttt{write}$. This event can not be in $\beta_n(\alpha)[j]$ if $WS_n(t,j') \cap WS_n(t',j'') \neq \emptyset$ because by a similar reasoning as in the case (c) above, $t' \in getConflicts(\langle t, info(t)\rangle)$. Thus, $\mathbf{submit}_n(t', op')$ happens after event $\nu_{j'}$ in $\beta_n(\alpha)$. There must exist the event $\mathbf{execute\_op}_n(t', op')$ after π_{z_1} in α for which $\mathbf{submit}_n(t', op')$ is part of its action. But as $ws\_run = \texttt{true}$ and $type(op') = \texttt{write}$, this event is disabled forever. Therefore, $\nu_{j''} \neq \mathbf{notify}_n(t', op', active)$. As Assumption 4.3 holds in $\beta_n(\alpha)$, $\nu_{j''} = \mathbf{notify}_n(t, info(t).WS, active)$ and thus, there exists $z_2 > z_1$ such that $\pi_{z_2} = \mathbf{notify}_n(t, info(t).WS, active)$. By its effects, $s_{z_2}.status_n(t) = active$. In conclusion, $\forall\, z' \geq z_2$, the event $\mathbf{end\_commit}_n(t)$ is enabled at $s_{z'}$. By weak-fairness of α, $\exists\, z' > z_2\colon \pi_{z'} = \mathbf{end\_commit}_n(t)$. Therefore, $s_{z'}.status_n(t) = committed$. The Lemma is verified in such a case.

□

It is also needed to show that all delivered messages are going to be processed. Some messages will fail in the certification process (where messages are discarded, see the $\mathbf{discard\_ws}_n$ action in Figure 3) while others not. By Lemma 1, those messages that have passed the certification phase will be eventually committed and the next message will be processed. Let us define a function $d(m, head(channel_n)) = k \in \mathbb{Z}^+$ that represents the distance between the position of a message m in $channel_n$ to its head, measured by the number of preceding messages. The next Property ensures that a message will eventually get processed provided that the site is a correct site; in terms of the new function $d(m, head(channel_n)) < k$ with k decreasing as more preceding messages are processed.

Property 14. *Let α be a fair execution of the RP and let s_z be a reachable state in α.*

$$d(m, head(s_z.channel_n)) = k > 0 \Rightarrow \exists\, z' > z\colon d(m, head(s_{z'}.channel_n)) < k \vee s_{z'}.site\_state_n = crashed.$$

Proof. By contradiction. We assume the property does not hold: $\forall\, z' \geq z\colon d(m, head(s_{z'}.channel_n)) \geq k \wedge s_{z'}.site\_state_n = alive$.
As $k > 0$, there exists $\langle t, info(t)\rangle = head(s_z.channel_n)$. We only need to prove that there is a $z' > z$ such that $s_{z'}.received_n$ includes $\langle t, info(t)\rangle$. In such a way, $d(m, head(s_{z'}.channel_n)) < k$.
At s_z the following holds:

- Case 1: $\neg certification(\langle t, info(t)\rangle, s_z.SEQ_n)$.
  The enabled condition of $\pi = \mathbf{discard\_ws}_n(t)$ is true at s_z. No other event can modify such a condition due the fact $\langle t, info(t)\rangle = head(s_z.channel_n)$ and π is the only event able to modify $channel_n$ in such a situation. As $\forall\, z' \geq z\colon s_{z'}.site\_state_n = alive$, π is enabled $\forall\, s_{z'}$. This is not possible because α is a fair execution. By weak fairness, there exists $\pi_{z'} = \mathbf{discard\_ws}_n(t)$ in α. By its effects, $s_{z'}.received_n$ includes $\langle t, info(t)\rangle$.

- Case 2: $certification(\langle t, info(t)\rangle, s_z.SEQ_n)$.
  By the conditions of Lemma 1 and the fact that $\forall\, z' \geq z\colon s_{z'}.site\_state_n = alive$, then $\exists\, z' > z\colon s_{z'}.status_n(t) =$

$committed$. By Property 7, $\exists\, z_1, z \le z_1 \le z' \colon \pi_{z_1} = \mathbf{end\_commit}_n(t)$. By its effects, $s_{z_1}.received_n$ includes $\langle t, info(t)\rangle$.

□

The next lemma states that if a transaction has committed at a given site, it will get eventually committed at the rest of correct sites. This lemma is the origin of the "*uniform commit*" property.

Lemma 2. *Let α be a fair execution of the RP and let s_z be a reachable state in α.*

$$s_z.status_n(t) = committed \Rightarrow \forall\, n' \in N \colon (\exists\, z' \colon s_{z'}.status_{n'}(t) = committed \vee s_{z'}.site\_state_{n'} = crashed)$$

Proof. By contradiction. We assume the Lemma does not hold; that is, $\exists\, n' \in N \colon (\forall\, z' \colon s_{z'}.status_{n'}(t) \neq committed \wedge s_{z'}.site\_state_{n'} = alive)$. In that case, $n' \in Correct(N)$.
Let s_z be the first state in α such that $s_z.status_n(t) = committed$. By Property 7, $\pi_z = \mathbf{end\_commit}_n(t)$. Its enabled condition at s_{z-1} verifies $s_{z-1}.status_n(t) = active$, and $\langle t, info(t)\rangle = head(s_{z-1}.channel_n)$. By Property 12, it also verifies $certification(\langle t, info(t)\rangle, s_z.SEQ_n) = \texttt{true}$. Notice that by its effects $\langle t, info(t)\rangle \in s_z.received_n$.
By Property 5 (following its notation), $\exists\, z_4, z_4 \le z-1 \colon \pi_{z_4} = \mathbf{deliver}_n(\langle t, info(t)\rangle)$. In the associated trace $\gamma(\alpha) \in Traces(GCS)$, the event $\mathbf{deliver}_n(\langle t, info(t)\rangle)$ appears in it. Thus by Assumption 5.7 (Uniform Agreement) and Assumption 5.4 (No Duplication), there exists in $\gamma(\alpha)$ a unique event $\mathbf{deliver}_{n'}(\langle t, info(t)\rangle)$ because $n' \in Correct(N)$. Therefore, in α there exists a unique $\pi_{z'} = \mathbf{deliver}_{n'}(\langle t, info(t)\rangle)$. By its effects, $s_{z'}.channel_{n'} \ni \langle t, info(t)\rangle$. By Property 14: $\exists\, z'', z'' > z' \colon \langle t, info(t)\rangle = head(s_{z''}.channel_{n'})$.
We consider the following cases taking into account Property 10 and the relation between z'' and $z-1$:

Case 1. $z'' > z-1$ and $s_{z''}.received_{n'} \preceq s_{z''}.received_n$.
Case 2. $z'' > z-1$ and $s_{z''}.received_n \preceq s_{z''}.received_{n'}$.
Case 3. $z'' \le z-1$ and $s_{z''}.received_{n'} \preceq s_{z''}.received_n$.
Case 4. $z'' \le z-1$ and $s_{z''}.received_n \preceq s_{z''}.received_{n'}$.

Only Case 2 is not possible because $\langle t, info(t)\rangle \in s_z.received_n$ and it is not included in $s_{z''}.received_{n'}$. In the rest of cases, by applying Property 11 it is concluded the following: $s_{z''}.SEQ_{n'} = s_{z-1}.SEQ_n \wedge s_{z''}.received_{n'} = s_{z-1}.received_n$. Therefore, $certification(\langle t, info(t)\rangle, s_{z''}.SEQ_{n'}) = certification(\langle t, info(t)\rangle, s_z.SEQ_n) = \texttt{true}$. The conditions of Lemma 1 hold: $\langle t, info(t)\rangle = head(s_{z''}.channel_{n'}) \wedge certification(\langle t, info(t)\rangle, s_{z''}.SEQ_{n'})$.
By Lemma 1, $\exists\, z''' > z'' \colon s_{z'''}.status_{n'}(t) = committed$. The Lemma is proved. □

Finally, the following theorem translates what it has been set up by Lemma 2 for the RP to the DBS_n. It is a must that for every committed transaction in the associated DBS_n of a site, it will eventually get committed in the remainder set of $DBS_{n'}$ with n' belonging to the set of $Correct(N)$.

Theorem 2 (Liveness - Uniform Commit)**.** *The RP verifies the liveness criterion: Uniform Commit.*

Proof. Let $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ be an arbitrary execution of the RP. If $s_z.status(t) = committed$ then $\exists\, z_1, z_1 \le z \colon \pi_{z_1} = \mathbf{end\_commit}_n(t)$. By Lemma 2, $\pi_z = \mathbf{end\_commit}_n(t) \Rightarrow \forall\, n', n' \in N \colon \exists\, z' \colon \pi_{z'} = \mathbf{end\_commit}_{n'}(t) \vee \pi_{z'} = \mathbf{crash}_{n'}$. In other words, if we narrow the second term for all correct sites: $\forall n', n' \in Correct(N) \colon \exists\, z' \colon \pi_{z'} = \mathbf{end\_commit}_{n'}(t)$.
Let $\beta_n(\alpha) \in Traces(DBS_n)$ and $\beta_{n'}(\alpha) \in Traces(DBS_{n'})$ be the associated traces of $Events(DBS_n)$ and $Events(DBS_{n'})$ respectively, with $n' \in Correct(N)$. By the effects of $\pi_{z_1} = \mathbf{end\_commit}_n(t)$, the $\mathbf{commit}_n(t)$ event is present in $\beta_n(\alpha)$ and, hence, this will imply that $\forall\, n', n' \in Correct(N) \colon \mathbf{commit}_{n'}(t)$ in $\beta_{n'}(\alpha)$. Hence, the Theorem is proved. □

8 Discussion

About the writeset extraction. First of all, we have not considered SQL statements throughout all the paper, we merely pointed out that we read (or write) a data item version ($\langle x, value\rangle$). Writeset extraction of a transaction is a well known problem in the replicated database community and of particular interest in middleware approaches due to the heterogeneity of DBS used. It can be achieved through different mechanisms like: storing SQL statements [24, 36]; using triggers to extract the changes [22]; and, reading changes from the DBS log [30]. Each one of these techniques presents its own advantages and drawbacks that will be outlined in the following. Thus, for example, in the first technique an inconsistent state can be reached with non-deterministic SQL statements, such as those setting up the current timestamp (among others) that will be different at each replica and must be re-written on the fly (or properly modified at the time the writeset is multicast) [40]. However, this technique is the most straightforward one as update statements are stored at the middleware layer of the transaction delegate replica as they are issued. Trigger based extraction is a rather standard approach. This alternative presents the problem of the overhead imposed at the DBS_n as more write operations are needed for each individual write operation performed by a SQL statement. Nevertheless, this approach avoids the inconsistencies across replicas present in the previous solution. Finally, the most attractive way to achieve this is by reading the log at the transaction delegate replica. Therefore, no overhead is incurred by this read operation. However, though each DBS offers its own interface for accessing the log, this is not an easy task (at least from a middleware approach point of view), since it highly depends on the kind of DBS considered and even version changes of the DBS may lead to a whole redefinition of this mechanism [25, 51]. All these aspects have thoroughly discussed from a practical point of view in [38].

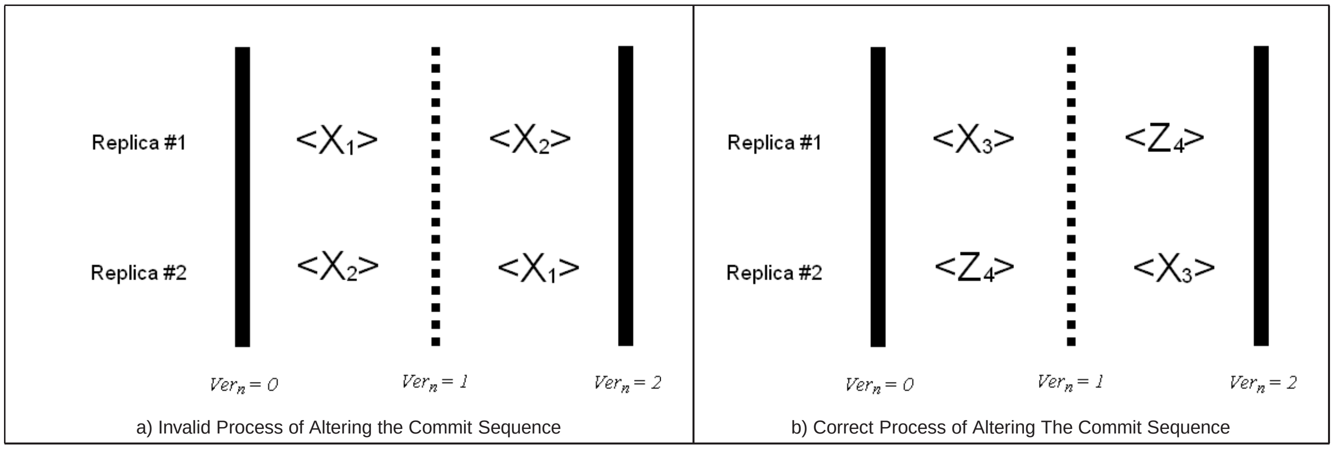

Figure 4: Altering the Commit Ordering at Two Different Replicas

Prefix-Order Consistency of the Sequencer Correctness Criterion. In order to ensure its correctness, RP needs that remote writeset application follows a sequential order –being such order identical in all sites– which has been shown in [19] as a sufficient condition for generating a GSI isolation level (provided that individual DBS_n are SI). This is a very strong requirement for generating a one-copy equivalence scheduler for transactions in a replicated database. Actually, transactions can be applied in any order under certain circumstances. Let us see this with an example, given four transactions: $\{T_1, T_2, T_3, T_4\}$ that respectively write data items: X_1, X_2, X_3, Z_4. T_1 and T_2 are executed in different order at two replicas, while in another system execution only T_3 and T_4 are executed in different order (as it can be seen in Figures 4.a and 4.b). From the execution depicted above, we can say that if the final snapshot of the altered sequence is the same then it will not matter whether they are committed or not in the same order (as it is shown in 4.b). This approach has been considered in [30] as "*holes*" appearing in the associated SEQ_i variable. Nevertheless, in order to maintain its correctness, the beginning of new local transactions should be avoided as long as there are holes in SEQ_i, since they will get an inconsistent snapshot for their read operations. A slight variation of this enhancement is to concurrently apply non-conflicting transactions and commit them in the order they were certified. The impact of this modification has not been experimentally tested yet.

Garbage Collection of the Sequencer and Fault-Tolerance Issues with the Crash-Recovery Model. Another closely related issue is the need of a sequencer variable to store the whole set of successfully certified transactions. This variable seems to infinitely grow along a system execution. A straightforward solution to avoid this is by piggybacking the number of the last applied writeset in the writesets regularly broadcast by each replica. Thus, the sequencer can be trimmed accordingly by removing until the minimum version received. This technique has been discussed in [50]. Of course, in case of a replica that crashes, the number of collected version numbers must be properly modified so the process of garbage collection of the sequencer may continue without interruptions. Sequencer usage has been shown as an interesting tool for the crash-recovery model in replicated databases [27, 44, 4]. Though this failure scenario has not been considered in this paper, it can be accomplished by considering the view synchrony properties featured by the GCS [13]. Most of these recovery solutions follow the same philosophy, once a replica fails the process of garbage collection of the sequencer is stopped. Hence, when the failed node rejoins the system, a recoverer replica must be chosen to perform the data transfer of the missing part of the sequence. A comparing evaluation of recovery techniques is given in [44].

Commitment of a Successfully Certified Writeset. Up to our knowledge, something that has not been carefully (*formally*) considered is the fact that after a writeset has been successfully certified, it must be committed. Nevertheless, there can exist several (local ones from RP's point of view) transactions that may conflict with the certified writeset. Hence, it may happen that the transaction applying a writeset may become blocked, or worse aborted. This last point was studied in [30]. They propose a time-out approach for blocking time and re-attempt the transaction just in case of time-out expiration or the transaction is rolled back by the DBS_n. Another alternative approach, presented in [33], is to keep track at the middleware layer of the metadata conflict table contained in the DBS_n so they can be aborted as soon as possible (i.e. before these transactions reach the $request\_commit_n(t)$ event with t being local at replica n). Moreover, both approaches do not prevent the fact that the writeset can be endlessly aborted by the DBS_n due to new write operations unless the replication protocol takes some more control actions. It needs to stop the execution of new write operations of currently executing transactions and the beginning of new transactions that may potentially write in the database. This last fact has been considered in the design and formalization of RP. Another approach, totally orthogonal to the previous ones presented here, is to define a conflict aware load balancing technique by way of conflict classes [35, 3, 52]. Under this assumption, though it is full replicated, clients have to pre-declare the set of data items about to be read and/or written by a transaction which is denoted as a conflict class. Each replica is the owner of one or several conflict classes. Moreover, a transaction can potentially access several conflict classes and, typically, it is statically assigned to one master. The goal here is to minimize inter-replica conflicts, which is left to the masters of the respective conflict classes, and the throughput of applying remote transactions is maximized. More precisely, transactions are broadcast (using the total-order primitive) to all replicas. Master sites execute transactions, while the rest just apply the resulting writesets in the order transactions were delivered. However, this last alternative presents the inconvenience of its high dependence of the database application considered: an evolution of the database schema may convert the workload distribution policy into a useless one. Finally, a second possible optimization is presented in [16] and it consists in uniting transaction ordering with transaction durability in a replicated setting. This may be achieved either at the middleware layer, or at the database one, generating respectively the Tashkent-MW and Tashkent-API systems described in that paper. In both cases, multiple writesets are grouped and committed at once. This also enhances throughput since many intermediate commits are eliminated, as it is usually done in isolated DBSs. Moreover, this optimization is also able to preserve the initial ordering without many problems.

Transaction Isolation Level Reached with the Prefix-Order Consistency in the Replicated Database with SI DBSs. The prefix-consistency order imposed by the RP and the usage of DBS_n providing SI have been shown as a sufficient condition to provide GSI [19] or *weak* SI in terms of [14]. RP does not support either *strong session* SI [14, 40] where clients always see their own updates. Unless they access all the time the same replica, which is not necessarily true in the case of a crash failure. Moreover, clients are not guaranteed to see increasing snapshots when the same client executes consecutive queries on different replicas that are not equally up-to-date. The same problem appears in [17, 30, 33]. This problem can be circumvented by including some meta-information on the client side about the latest version seen by him, upon reception of a query the replica can check its current database version and may decide to execute the query if the proper version is given, forward it to another replica or block the transaction until the proper snapshot version of the client is reached. It has been shown in [17, 19] that there is no non-blocking implementation that provides conventional (or strong [14]) SI. Though this fact can be taken as a chance to consider data freshness in two ways: either to obtain a given snapshot by selecting the proper replica that holds the requested snapshot *version* (or *timestamp*) [41] or trying to define an interval of validity of a given snapshot, such as *k*-bound GSI [6] where a $start$ message (containing the snapshot *version*) is broadcast (using total-order) at the beginning of the transaction. Upon its reception it is checked against the last committed transaction if its difference is greater than *k* which is a parameter chosen by the database application.

Practical Considerations with Commercial DBSs that provide Different Isolation Levels. The certification process proposed here is a distributed and replicated version of the *first-committer-wins* (FCW) rule of SI [7]. Note that most of DBS_n commercial implementations (such as PostgreSQL [42] and Oracle [34]) use another different flavor of this rule called *first-updater-wins* (FUW) rule. Only the first transaction performing a write operation over the same data item with other update transactions is allowed to proceed while the others remain blocked. At the moment when the former one commits the rest will be aborted. Recall the conflict detection mechanism with local transactions while applying a remote writeset that we have previously discussed [33] that becomes specially useful with a DBS_n using the FUW though correctness is not affected either by using a DBS_n with a FUW or a FCW rule. Read-Committed (RC) is a weaker isolation level rather than the one provided by SI [7]. Actually, it is the default isolation level provided by PostgreSQL and Oracle. For each read operation executed by a transaction executed under this isolation level, it is performed over the current database snapshot version at the moment it is issued (i.e. non-repeatable reads are possible [7]). Whereas the commitment rule for a transaction under RC is done by relaxing the FUW rule, whenever the transaction holding the update lock on a data item commits, the rest of blocked transactions are re-evaluated but they do not get aborted as opposed to FUW under SI. Thus, if we want to run RP with all DBS_n providing RC there is no need for a sequencer, the certification of a transaction is done by way of the total-order delivery of the writeset by the GCS. Of course, it is still needed to control the execution flow of local transactions as in GSI has been done and will imply some abortions of local transactions, reaching a *Generalized* RC isolation level for the replicated database [8]. Moreover, the certification process was originally proposed to obtain One-Copy-Serializability (1CS) [10, 36] with each DBS_n providing the serializable isolation level. Again, RP can provide 1CS if it propagates the readsets along with the writesets in the message being broadcast, as it has been used in [35, 28, 36]. 1CS can also be reached with the usage of DBS_n featuring SI either: statically (i.e. forcing transactions to generate histories where there are no intersection between readsets and writeset) [18]; or, dynamically by shifting read operations into write operations [17] (a "`SELECT`" SQL statement can be modified into a "`SELECT ... FOR SHARE`" or a "`SELECT ... FOR UPDATE`" statement).

Integrity Constraints on Replicated Databases. One topic that has not been covered yet, to the best of our knowledge, is integrity constraints (IC) at replicated databases. At this point a transaction can be aborted by an IC operation (i.e. it may obtain a $notify_n(t, op, result, data)$ with $result = abort$ due to an IC restriction violation). Furthermore, a transaction applying a remote writeset may be aborted due to this fact too. One possible way to overcome this is to serially execute transactions that execute IC operations [1]. Thus, the message containing the writeset for its certification must also include all data items that have been involved in IC operations and, thanks to this, observe all IC in the replicated database. Of course, there are some possible optimizations in the certification function, though this was not one of the scopes of the paper, such as an IC-read operation over the same data item between two or more concurrent transactions is totally compatible.

Alternative Approaches for the Correctness Proof. The first approach of setting up a correctness criterion for replicated data was formally stated in [10]. It was established the serialization theory for a replicated system by giving the notion of one copy equivalence that derived, by the usage of 2PL DBS replicas, to the 1CS correctness criterion. A given solution is said to be 1CS if its associated serialization graph is acyclic [10]. This has been commonly used in previous works in order to prove the validity of the solutions proposed. Therefore, the solutions proposed in [28, 36, 24, 35] share the same characteristic: transactions are executed at their delegate replicas and writesets are broadcast, using the total-order facility, to the rest of replicas and differ in the way transactions are terminated (either weak-voting or distributed certification [50]) to guarantee they are 1CS. The total-order (that can also be first *optimistically* [28, 35] delivered) delivery of writesets ensures that the way the serialization graph is built prevents the apparition of any cycle. A good example of this is the solution presented in [28] for a kernel-based and [35] for a middleware architecture where the whole transaction (e.g. defined as a stored procedure pretty much like a web data-sheet form) is broadcast. Nevertheless, most of commercial DBS do provide SI and several attempts [17, 30, 19] have been done to provide conditions and define a correctness criterion pretty similar to the notion of 1CS. Therefore, in [19, 17] the notion of prefix-order SI consistency is defined which is considered as a sufficient condition to provide SI histories [19] as we have depicted above, i.e. it disallows some histories that are also valid SI histories. A centralized and a distributed certification algorithm are presented in [17] based on the previous concept and transactions obtain GSI. The correctness proof for the distributed certification (which is somehow equivalent to our RP) is based on the total order delivery of messages containing the writeset [17]. Concurrently to [17], an equivalent notion for SI replicas is given in [30] which is called 1-copy-SI stating that a set of replicas provides it if there is a ROWA [20] function that maps how transactions are executed at all replicas and there exists a SI history that respects the commit ordering of write-write conflicting transactions and the start ordering of write-read conflicting transactions. They propose a distributed certification algorithm for a middleware architecture that is based on the sequential execution of writesets that are total-order delivered. Therefore, it is easy to show that it follows the same approach as in [17]. To the best of our knowledge, none of the previous works, either for 1CS [28, 36, 24, 35] or GSI [17, 30], have used a typical approach for verifying the correctness of distributed systems such as TLA+ [29], the state transition systems [46] or the Input/Output automata theory [32]. Moreover, 1CS solutions present new algorithms, with independent specifications, analysis and correctness proofs, though the same can not be said for GSI protocols where the correctness criterion and algorithms are introduced in the same whole unit[17, 30].

Moreover, all previous algorithms proposals base their data consistency on the total order delivery facility provided by a GCS though they do not cope with the crash failure of a replica and how it affects its correctness. In our work, we have formally introduced a replication protocol for a database replication middleware architecture with SI replicas as a state transition system that deals with crash failures which is a novel approach. More recently, a formalization of the deferred update technique for database replication with serializable databases is proposed in [45] where some characteristics and limitations of such technique are presented. They set up that the termination protocol must totally order globally committed transactions, similar to the approach presented in our work but this is a sufficient condition for SI. They have also shown that it is only needed to preserve the serializability order only for those transactions that modify the database without taking into account read-only transactions, also similar to the prefix-consistency correctness criterion presented here and in [19, 17] for SI. Finally, they have checked their specifications using the TLA+ model checker which is also a novel approach since they use distributed system tools to verify the correctness of database replication protocols.

9 Conclusions

In this paper, it has been formalized a certification-based database replication protocol (RP) as a state transition system. This represents a novel approach, since none of the previous approaches has used distributed-system-specific tools to ensure its correctness. Moreover, they have introduced their specifications of replication protocols using independent specifications, analysis and correctness proofs. We have established safety (*database prefix order consistency*: for each pair of replicas the database of one of them constitutes the prefix of the other or viceversa) and liveness correctness (*uniform commit*: if a site commits a transaction then every site will commit this transaction or has crashed) criteria that needs to be verified by RP.

RP has been introduced in a middleware architecture which is composed by the replication protocol itself, a set of database replicas (DBS_n) and a group communication system (GCS). Hence, our formalization has to introduce a composition rule of the different components in order to define the executions of the system. RP represents a very basic version of a certification-based replication protocol, in the sense that it is an eager update everywhere replication protocol where transactions are sequentially certified and committed thanks to the total order delivery of writesets performed by the GCS. Our main aim is to ensure its correct behavior (i.e. database prefix order consistency and uniform commit) using a state transition system even in the presence of crash failures, something that, to the best of our knowledge, has been missed in previous proposals. Transactions executed with RP, as shown in [19, 17], obtain GSI provided that each DBS_n used is SI.

Finally, it has been discussed the limitations of RP and proposed several optimizations, many of them already included in several works. Another phenomenon that was not discussed in-depth yet is the way a writeset is applied at a remote replica and how it deals with possible conflicts between local transactions. We also provide an extension of the replication protocol to cope with growth of the sequencer and the crash-recovery model. The alternative approaches of correctness criteria used in other works have been depicted and discussed too.

Acknowledgments

This work has been supported by FEDER and the Spanish Government under research grant TIC2006-14738-C02.

References

[1] Atul Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, Massachusetts Institute of Technology, March 1999.

[2] Divyakant Agrawal, Gustavo Alonso, Amr El Abbadi, and Ioana Stanoi. Exploiting atomic broadcast in replicated databases (extended abstract). In Christian Lengauer, Martin Griebl, and Sergei Gorlatch, editors, *Euro-Par*, volume 1300 of *Lecture Notes in Computer Science*, pages 496–503. Springer, 1997.

[3] Cristiana Amza, Alan L. Cox, and Willy Zwaenepoel. Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites. In Markus Endler and Douglas C. Schmidt, editors, *Middleware*, volume 2672 of *Lecture Notes in Computer Science*, pages 282–304. Springer, 2003.

[4] José Enrique Armendáriz-Iñigo, José Ramón Garitagoitia, Francesc D. Muñoz-Escoí, and José Ramón González de Mendívil. MADIS-SI: A database replication protocol with easy recovery. Technical Report ITI-ITE-06/05, Instituto Tecnológico de Informática, June 2006.

[5] José Enrique Armendáriz-Iñigo, José Ramón González de Mendívil, and Francesc D. Muñoz-Escoí. A lock-based algorithm for concurrency control and recovery in a middleware replication software architecture. In *HICSS*, page 291a. IEEE Computer Science, 2005.

[6] José Enrique Armendáriz-Iñigo, José Ramón Juárez-Rodríguez, José Ramón González de Mendívil, Hendrik Decker, and Francesc D. Muñoz-Escoí. *K*-bound GSI: a flexible database replication protocol. In Yookun Cho, Roger L. Wainwright, Hisham Haddad, Sung Y. Shin, and Yong Wan Koo, editors, *SAC*, pages 556–560. ACM, 2007.

[7] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. A critique of ANSI SQL isolation levels. In Michael J. Carey and Donovan A. Schneider, editors, *SIGMOD Conference*, pages 1–10. ACM Press, 1995.

[8] Josep M. Bernabé-Gisbert, Raúl Salinas-Monteagudo, Luis Irún-Briz, and Francesc D. Muñoz-Escoí. Managing multiple isolation levels in middleware database replication protocols. In Minyi Guo, Laurence Tianruo Yang, Beniamino Di Martino, Hans P. Zima, Jack Dongarra, and Feilong Tang, editors, *ISPA*, volume 4330 of *Lecture Notes in Computer Science*, pages 511–523. Springer, 2006.

[9] Philip A. Bernstein. Middleware: A model for distributed system services. *Commun. ACM*, 39(2):86–98, 1996.

[10] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[11] Michael J. Carey and Miron Livny. Conflict detection tradeoffs for replicated data. *ACM Trans. Database Syst.*, 16(4):703–746, 1991.

[12] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. C-JDBC: Flexible database clustering middleware. In *USENIX Annual Technical Conference, FREENIX Track*, pages 9–18. USENIX, 2004.

[13] Gregory Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, 2001.

[14] Khuzaima Daudjee and Kenneth Salem. Lazy database replication with snapshot isolation. In *VLDB*, Seoul, Korea, September 2006.

[15] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[16] Sameh Elnikety, Steven Dropsho, and Fernando Pedone. Tashkent: Uniting durability with transaction ordering for high-performance scalable database replication. In *ACM Eurosys*, Leuven, Belgium, April 2006.

[17] Sameh Elnikety, Fernando Pedone, and Willy Zwaenopoel. Database replication using generalized snapshot isolation. In *SRDS*. IEEE Computer Society, 2005.

[18] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005.

[19] José Ramón González de Mendívil, José Enrique Armendáriz-Iñigo, Francesc D. Muñoz-Escoí, Luis Irún-Briz, José Ramón Garitagoitia, and José Ramón Juárez-Rodríguez. Non-blocking ROWA protocols implement GSI using SI replicas. Technical Report ITI-ITE-07/10, Instituto Tecnológico de Informática, May 2007.

[20] Jim Gray, Pat Helland, Patrick E. O'Neil, and Dennis Shasha. The dangers of replication and a solution. In H. V. Jagadish and Inderpal Singh Mumick, editors, *SIGMOD Conference*, pages 173–182. ACM Press, 1996.

[21] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Dep. of Computer Science, Cornell University, Ithaca, New York (USA), May 1994.

[22] Luis Irún-Briz, Hendrik Decker, Rubén de Juan-Marín, Francisco Castro-Company, Jose E. Armendáriz-Iñigo, and Francesc D. Muñoz-Escoí. MADIS: A slim middleware for database replication. In José C. Cunha and Pedro D. Medeiros, editors, *Euro-Par*, volume 3648 of *Lecture Notes in Computer Science*, pages 349–359. Springer, 2005.

[23] José Ramón Juárez-Rodríguez, José Enrique Armendáriz-Iñigo, José Ramón González de Mendívil, Francesc D. Muñoz-Escoí, and José Ramón Garitagoitia. A weak voting database replication protocol providing different isolation levels. In *NOTERE'07*, 2007.

[24] Bettina Kemme. *Database Replication for Clusters of Workstations (ETH Nr. 13864)*. PhD thesis, Swiss Federal Institute of Technology, Zurich, Switzerland, 2000.

[25] Bettina Kemme and Gustavo Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang, editors, *VLDB*, pages 134–143. Morgan Kaufmann, 2000.

[26] Bettina Kemme and Gustavo Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Trans. Database Syst.*, 25(3):333–379, 2000.

[27] Bettina Kemme, Alberto Bartoli, and Özalp Babaoglu. Online reconfiguration in replicated databases based on group communication. In *DSN*, pages 117–130. IEEE-CS Press, 2001.

[28] Bettina Kemme, Fernando Pedone, Gustavo Alonso, André Schiper, and Matthias Wiesmann. Using optimistic atomic broadcast in transaction processing systems. *IEEE Trans. Knowl. Data Eng.*, 15(4):1018–1032, 2003.

[29] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison Wesley Professional, 2002.

[30] Yi Lin, Bettina Kemme, Marta Patiño-Martínez, and Ricardo Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *SIGMOD Conference*, 2005.

[31] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *PODC*, pages 137–151, 1987.

[32] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. Technical Report MIT/LCS/TM-373, Massachusetts Institute of Technology, 1988.

[33] Francesc D. Muñoz-Escoí, Jerónimo Pla-Civera, María Idoia Ruiz-Fuertes, Luis Irún-Briz, Hendrik Decker, José Enrique Armendáriz-Iñigo, and José Ramón González de Mendívil. Managing transaction conflicts in middleware-based database replication architectures. In *SRDS*, pages 401–410, 2006.

[34] Oracle Corporation. Oracle *11g* Release 1. Accessible in URL: `http://download.oracle.com/docs/cd/B28359_01/server.111/b28318.pdf`, 2007.

[35] Marta Patiño-Martínez, Ricardo Jiménez-Peris, Bettina Kemme, and Gustavo Alonso. MIDDLE-R: Consistent database replication at the middleware level. *ACM Trans. Comput. Syst.*, 23(4):375–423, 2005.

[36] Fernando Pedone. *The database state machine and group communication issues (Thèse N. 2090)*. PhD thesis, École Polytecnique Fédérale de Lausanne, Lausanne, Switzerland, 1999.

[37] Fernando Pedone, Rachid Guerraoui, and André Schiper. Exploiting atomic broadcast in replicated databases. In *Euro-Par*, 1998.

[38] Christian Plattner. *Ganymed: A Platform for Database Replication (ETH Nr. 16945)*. PhD thesis, Swiss Federal Institute of Technology, Zurich, Switzerland, 2006.

[39] Christian Plattner and Gustavo Alonso. Ganymed: Scalable replication for transactional web applications. In Hans-Arno Jacobsen, editor, *Middleware*, volume 3231 of *Lecture Notes in Computer Science*, pages 155–174. Springer, 2004.

[40] Christian Plattner, Gustavo Alonso, and M. Tamer-Özsu. Extending DBMSs with satellite databases. *The VLDB Journal*, 2006.

[41] Christian Plattner, Andreas Wapf, and Gustavo Alonso. Searching in time. In Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis, editors, *SIGMOD Conference*, pages 754–756. ACM, 2006.

[42] PostgreSQL. The world's most advance open source database web site. Accessible in URL: `http://www.postgresql.org`, 2007.

[43] Luís Rodrigues, Hugo Miranda, Ricardo Almeida, João Martins, and Pedro Vicente. The GlobData fault-tolerant replicated distributed object database. In *EurAsia-ICT*, pages 426–433, 2002.

[44] María Idoia Ruiz-Fuertes, Jerónimo Pla-Civera, José Enrique Armendáriz-Iñigo, José Ramón González de Mendívil, and Francesc D. Muñoz-Escoí. Revisiting certification-based replicated database recovery. In Robert Meersman and Zahir Tari, editors, *OTM Conferences (1)*, volume 4803 of *Lecture Notes in Computer Science*, pages 489–504. Springer, 2007.

[45] Rodrigo Schmidt and Fernando Pedone. A formal analysis of the deferred update technique. Technical Report LABOS-REPORT-2007-002, École Polytechnique Fédérale de Lausanne (EPFL), 2007.

[46] A. Udaya Shankar. An introduction to assertional reasoning for concurrent systems. *ACM Comput. Surv.*, 25(3):225–262, 1993.

[47] Michael Stonebraker. Concurrency control and consistency of multiple copies of data in distributed ingres. *IEEE Trans. Software Eng.*, 5(3):188–194, 1979.

[48] Sybase, Inc. Replication strategies: Data migration, distribution and synchronization. White paper, November 2003. 30 pages.

[49] Matthias Wiesmann, Fernando Pedone, André Schiper, Bettina Kemme, and Gustavo Alonso. Understanding replication in databases and distributed systems. In *ICDCS*, pages 464–474, 2000.

[50] Matthias Wiesmann and André Schiper. Comparison of database replication techniques based on total order broadcast. *IEEE TKDE*, 17(4):551–566, April 2005.

[51] Shuqing Wu and Bettina Kemme. Postgres-R(SI): Combining replica control with concurrency control based on snapshot isolation. In *ICDE*, pages 422–433. IEEE Computer Society, 2005.

[52] Vaide Zuikeviciute and Fernando Pedone. Conflict aware load balancing techniques for database replication. In *23rd ACM Symposium on Applied Computing (ACM SAC 2008)*. ACM, 2008. Accepted for Publication.

A Appendix

A.1 Proof of Property 1

Proof. Let s_z be the first state being $s_z.status_n(t) = active$. The only action making $s_z.status_n(t) = active$ is $\pi_z = \mathbf{notify}_n(t, op, active)$ for some $op \in OP$. □

A.2 Proof of Property 2

Proof. If $\pi_z = \mathbf{crash}_n$, then $s_z.site\_state_n = crashed$. No event of site $n \in N$ is enabled by the RP, and as $\pi_z \in Events(DBS_n)$, the Assumption 1.1 is preserved.

In order to prove the rest of Assumption 1, we have to study the following cases:

- Let $\nu_j = \mathbf{begin}_n(t)$ be an event in $\beta_n(\alpha)$, then there exists $\pi_z \in \{\mathbf{execute\_op}_n(t, op), \mathbf{execute\_ws}_n(t) \mid op \in OP\}$ in α such that ν_j is part of its action. The enabling condition for π_z requires that $s_{z-1}.status_n(t) = idle$. No other event $\pi_{z'}$, $z' < z$ is enabled being t its parameter. By its effects, $\forall\, z' > z-1 \colon s_{z'}.status_n(t) \neq idle$. Therefore, $\forall\, i < j \colon t \neq tran(\nu_i)$ in $\beta_n(\alpha)$, and Assumption 1.2 holds.

- Let $\nu_j \in \{\mathbf{commit}_n(t), \mathbf{abort}_n(t), \mathbf{notify}_n(t, op, abort)\}$ in $\beta_n(\alpha)$, then there exists $\pi_z \in \{\mathbf{end\_commit}_n(t), \mathbf{execute\_ws}_n(t'), \mathbf{notify}_n(t, op, abort) \mid t' \in T\}$ in α such that ν_j is part of its action. The actions of such events make $s_z.status_n(t) \in \{committed, aborted\}$. Thus, no other event $\pi_{z'}$, $z' \geq z$ is enabled being t its parameter. $\forall\, i > j \colon t \neq tran(\nu_i)$ and Assumption 1.3 holds.

- Let $\nu_j = \mathbf{submit}_n(t, op)$ in $\beta_n(\alpha)$, then there exists $\pi_z \in \{\mathbf{execute\_op}_n(t, op), \mathbf{execute\_ws}_n(t)\}$ in α such that ν_j is part of its action. If $s_{z-1}.status_n(t) = idle$ then the previous event for t in $\beta_n(\alpha)$ is $\mathbf{begin}_n(t)$ as the respective action indicates. If $s_{z-1}.status_n(t) = active$, the previous action for π_z in α which is the only one making $s_{z-1}.status_n(t) = active$ is $\mathbf{notify}_n(t, op', active)$ for some $op' \in OP$ as Property 1 indicates[6]. Thus, $\exists\, i < j \colon prev\_event(i, j, t) \wedge \nu_i \in \{\mathbf{begin}_n(t), \mathbf{notify}_n(t, op', active) \mid op' \in OP\}$ and Assumption 1.4 holds.

- Let $\nu_j = \mathbf{notify}_n(t, op, result)$ in $\beta_n(\alpha)$. This event is under control of the DBS_n component which is used to communicate to the RP the result of an operation submitted to the DBS_n. The RP guarantees that its enabling condition is true. Thus, the Assumption 1.5 is preserved.

- Let $\nu_j = \mathbf{commit}_n(t)$ in $\beta_n(\alpha)$, then there exists $\pi_z = \mathbf{end\_commit}_n(t)$ in α such that ν_j is part of its action. By Property 1, $\exists\, z_1 \leq z \colon \pi_{z_1} = \mathbf{notify}_n(t, op, active)$ for some $op \in OP$ in α. Let π_{z_1} be the last one, then $\exists\, i < j \colon prev\_event(i, j, t) \wedge \nu_i \in \{\mathbf{notify}_n(t, op, active) \mid op \in OP\}$ and Assumption 1.6 holds.

$\square$

A.3 Proof of Property 3

Proof. If $\pi_z = \mathbf{crash}_n$, then $s_z.site\_state_n = crashed$. No event of site $n \in N$ is enabled by the RP automaton, and as $\pi_z \in Events(GCS)$, the Assumption 5.1 is preserved.
In order to prove that Assumption 5.2 is preserved, we consider it does not hold: $\nu_i = \mathbf{broadcast}_n(m) \wedge \nu_j = \mathbf{broadcast}_{n'}(m) \Rightarrow i \neq j$ in $\gamma(\alpha)$. The only event in the RP that is able to broadcast a message is $\mathbf{request\_commit}_n(t)$ with $t.site = n$. Thus, there exist in α the events $\pi_z = \mathbf{request\_commit}_n(t)$ and $\pi_{z'} = \mathbf{request\_commit}_{n'}(t)$ such that ν_i and ν_j are part of their actions respectively being $m = \langle t, info(t)\rangle$. In the case of $n \neq n'$, $t.site \neq t.site$ is a contradiction. In the case of $n = n'$, if $z < z'$, then $s_z.sent_n(t) = \texttt{true}$ and $enabled(\pi_{z'})$ is false. $\square$

A.4 Proof of Property 4

Proof. By induction over the length of α. One can note that $\beta_n(\alpha[z]) = \beta_n(\alpha)[j]$. We have chosen the second notation in behalf of keeping the correctness proof simpler. In the next, $\beta_n(\alpha[0]) = \beta_n(\alpha)[0] = empty$. If $(s_z, \pi_{z+1}, s_{z+1})$ is a step of the RP, $\beta_n(\alpha[z+1]) = \beta_n(\alpha)[z] \cdot \nu_{j+1} \cdot \nu_{j+2} \cdot \cdots \cdot \nu_{j+k}$, where $\nu_{j+1} \ldots \nu_{j+k}$ are the events of $Events(DBS_n)$ executed by the event π_{z+1} of the RP as part of its action. Thus, we also write $\beta_n(\alpha[z+1]) = \beta_n(\alpha)[j+k]$.

- *Basis*. Let $\alpha = s_0$ be the initial state. For all $t \in T$ and $n \in N$: $s_0.status_n(t) = idle$, $s_0.site\_state_n = alive$, $s_0.info_n(t) = (0, \emptyset, 0)$, $s_0.Ver_n = 0$, and $s_0.SEQ_n = empty$. The Property holds at s_0.
  The associated trace is $\beta_n(\alpha)[0] = empty$. By definitions provided in Section 3: (Notice that $T(\beta_n(\alpha)[0]) = \emptyset$) For all $t \in T$, $idle_n(t, 0)$, $WS(t, 0) = \emptyset$; and $log(\beta_n(\alpha)[0]) = empty$.

- *Hypothesis*. Assume the Property 4 holds at s_z; and let $\beta_n(\alpha)[j]$ be its associated trace.

- *Induction Step*. Let $(s_z, \pi_{z+1}, s_{z+1})$ be a transition of the RP. We study how each possible π_{z+1} event affects the property; we only consider the events modifying the variables the property states.

[6] Note that there can be other $\pi_{z'}$ events that maintain the $active$ status for a given transaction, concretely the $\mathbf{request\_commit}_n(t)$ event, but such events do not extend the $\beta_n(\alpha)$ trace. In the last item will happen the same.

– $\pi_{z+1} = \mathbf{execute\_op}_n(t, op)$ with $t.site = n$.
By its enabled condition: $s_z.site\_state_n = alive$ and (a) $s_z.status_n(t) = idle$ or (b) $s_z.status_n(t) = active$. Proving the case (a) is sufficient because the action for (b) is included in (a). In that case: $\beta_n(\alpha)[j+2] = \beta_n(\alpha)[j] \cdot \mathbf{begin}_n(t) \cdot \mathbf{submit}_n(t, op)$. In the next obtained state by the action:
$s_{z+1}.status_n(t) = blocked$. By Definition 1, $blocked(t, j+2)$ and Property 4.3 holds.
$s_{z+1}.info_n(t).start = s_z.Ver_n$. By Hypothesis and Definition 4 (*log*), $s_z.Ver_n = |log(\beta_n(\alpha)[j])| = |log(\beta_n(\alpha)[j+1])| \wedge \nu_{j+1} = begin_n(t)$. No other event will modify $info_n(t).start$. Property 4.6 holds.
$s_{z+1}.info_n(t).WS = s_z.info_n(t).WS \cup \{op\}$ if $type(op) = \texttt{write}$. By Hypothesis and Definition 2 (writeset) $s_{z+1}.info_n(t).WS = WS(t,j) \cup \{op\} = WS(t, j+2)$. Property 4.7 holds.
The rest of cases in Property 4 hold trivially.

– $\pi_{z+1} = \mathbf{notify}_n(t, op, result)$.
In this case $\beta_n(\alpha)[j+1] = \beta_n(\alpha)[j] \cdot \mathbf{notify}_n(t, op, result)$. If $s_{z+1}.status_n(t) = active$ then by Definition 1 of transaction states $active(t, j+1)$ and Property 4.2 holds. If $s_{z+1}.status_n(t) = aborted$ then by Definition 1 of transaction states $aborted(t, j+1)$ and Property 4.5 holds. The rest of cases in Property 4 hold trivially.

– $\pi_{z+1} = \mathbf{crash}_n$.
In this case, $\beta_n(\alpha)[j+1] = \beta_n(\alpha)[j] \cdot \mathbf{crash}_n$. By the effects of the action, $s_{z+1}.status_n(t) = aborted$ if $s_z.status_n(t) \in \{blocked, active\}$. By Definition 1, $aborted(t, j+1)$ and Property 4.5 holds. Also $s_{z+1}.site\_state_n = crashed$ and the rest of cases in Property 4 hold.

– $\pi_{z+1} = \mathbf{execute\_ws}_n(t)$ with $t.site \neq n$.
In this case $\beta_n(\alpha)[j+k] = \beta_n(\alpha)[j] \cdot \mathbf{abort}_n(t_1) \cdot \dots \cdot \mathbf{abort}_n(t_{k-2}) \cdot \mathbf{begin}_n(t) \cdot \mathbf{submit}_n(t, s_{z+1}.info_n(t).WS)$ being $t_1, \dots, t_{k-2} \in getConflicts(\langle t, info(t) \rangle)$.
$s_{z+1}.status_n(t_i) = aborted, i\colon 1..k-2$ then by Definition 1 of transaction states $aborted(t, j+k)$ and Property 4.5 holds.
$s_{z+1}.status_n(t) = blocked$ then by Definition 1 of transaction states $blocked(t, j+k)$ and Property 4.3 holds.
As $s_{z+1}.info_n(t).WS = WS_n(t, j+k)$, by Definition 2, Property 4.7 holds.
The rest of cases in Property 4 trivially hold.

– $\pi_{z+1} = \mathbf{end\_commit}_n(t)$.
In this case $\beta_n(\alpha)[j+1] = \beta_n(\alpha)[j] \cdot \mathbf{commit}_n(t)$. By Definition 4, $log(\beta_n(\alpha)[j+1]) = log(\beta_n(\alpha)[j]) \cdot \langle t, WS(t,j), |log(\beta_n(\alpha)[j+1])| \rangle$. We study the effects of the event over the next state s_{z+1}.
$s_{z+1}.status_n(t) = committed$ and by Definition 1 of transaction states $committed(t, j+1)$. Property 4.4 holds.
$s_{z+1}.Ver_n = s_z.Ver_n + 1$. By Hypothesis $s_z.Ver_n + 1 = |log(\beta_n(\alpha)[j])| + 1 = |log(\beta_n(\alpha)[j+1])|$. Property 4.9 holds.
$s_{z+1}.info_n(t).end = s_{z+1}.Ver_n = |log(\beta_n(\alpha)[j+1])|$ and $\nu_{j+1} = \mathbf{commit}_n(t)$. Then Property 4.8 holds due to the fact that no other event can modify the variable $info_n(t).end$.
$s_{z+1}.SEQ_n = s_z.SEQ_n \cdot \langle t, s_{z+1}.info_n(t) \rangle$. Thus, $s_{z+1}.LOG_n = s_z.LOG_n \cdot \langle t, s_{z+1}.info_n(t).WS, s_{z+1}.info_n(t).end \rangle$.
By Hypothesis and the value of $s_{z+1}.info_n(t).end$:
$s_{z+1}.LOG_n = log(\beta_n(\alpha)[j]) \cdot \langle t, s_{z+1}.info_n(t).WS, |log(\beta_n(\alpha)[j+1])| \rangle$.
We only need to prove that $s_{z+1}.info_n(t).WS = WS(t,j)$.
In this case $s_{z+1}.info_n(t).WS = s_z.info_n(t).WS = WS_n(t,j)$ by induction Hypothesis (Property 4.7), and the fact that $s_z.site\_state_n = alive$, $\mathbf{end\_commit}_n(t)$ does not modify $s_z.info_n(t).WS$, and $\nu_{j+1} = \mathbf{commit}_n(t)$ does not modify $WS_n(t,j)$.
Therefore, $s_{z+1}.LOG_n = log(\beta_n(\alpha)[j]) \cdot \langle t, WS_n(t,j), |log(\beta_n(\alpha)[j+1])| \rangle = log(\beta_n(\alpha)[j+1])$, and Property 4.10 holds.
The rest of cases in Property 4 hold.

□

A.5 Proof of Property 9

Proof. By induction over the length of α.

- *Basis.* At $\alpha = s_0$, $\gamma(\alpha[0]) = empty$. Thus, $s_0.delivered_n = s_0.received_n = s_0.channel_n = mess_n(\gamma(\alpha[0])) = empty$. The property is verified at the initial state.

- *Hypothesis*. Assume the Property 9 is verified at s_z.
- *Induction step*. Let $(s_z, \pi_{z+1}, s_{z+1})$ be a transition step of the RP. The events affecting the variables of the property are $\pi_{z+1} \in \{\mathbf{deliver}_n(m), \mathbf{end\_commit}_n(t), \mathbf{discard\_ws}_n(t), \mathbf{crash}_n \mid t \in T, m \in M\}$.
  - $\pi_{z+1} = \mathbf{deliver}_n(m)$.
    In the associated trace, $\gamma(\alpha[z+1]) = \gamma(\alpha[z]) \cdot \mathbf{deliver}_n(m)$; and, by Definition 5, $mess_n(\gamma(\alpha[z+1])) = mess_n(\gamma(\alpha[z]))\cdot m$. By the effects of π_{z+1}, $s_{z+1}.delivered_n = s_z.delivered_n \cdot m$, and $s_{z+1}.channel_n = s_z.channel_n \cdot m$. By induction Hypothesis, the property holds.
  - $\pi_{z+1} \in \{\mathbf{end\_commit}_n(t), \mathbf{discard\_ws}_n(t) \mid t \in T\}$.
    In the associated trace, $\gamma(\alpha[z+1]) = \gamma(\alpha[z])$. In both cases, $\langle t, info(t)\rangle = head(s_z.channel_n)$. By their effects, $s_{z+1}.delivered_n = s_z.delivered_n$, $s_{z+1}.received_n = s_z.received_n \cdot \langle t, info(t)\rangle$, and $s_{z+1}.channel_n = tail(s_z.channel_n)$. Thus, $s_{z+1}.received_n \cdot s_{z+1}.channel_n = s_z.received_n \cdot s_z.channel_n$. By induction Hypothesis the property holds.
  - $\pi_{z+1} = \mathbf{crash}_n$. By its effects $s_{z+1}.channel_n = empty$, the rest of variables are not modified. By Hypothesis the property holds.

□

A.6 Proof of Property 12

Proof. We consider the following cases. The proof is made by contradiction.

- *Case 1*: $t.site \neq n$. By Property 1, $\exists\, z_1 \leq z \colon \pi_{z_1} = \mathbf{notify}_n(t, op, active)$. Then $\exists\, z_2 < z_1 \colon \pi_{z_2} = \mathbf{execute\_ws}_n(t)$, this is the event making $\mathbf{submit}_n(t, op)$ at s_{z_2} (Assumption 1.5). Thus, s_{z_2-1} verifies $\langle t, info(t)\rangle = head(s_{z_2-1}.channel_n)$ and $certification(t, s_{z_2-1}.SEQ_n)$. The only action that is able to modify $s_{z_2-1}.channel_n$ and $s_{z_2-1}.SEQ_n$ is $\pi_{z_3} = \mathbf{end\_commit}_n(t)$, being $z_2 < z_1 < z_3$. π_{z_3} has not been executed, accordingly $s_{z_2-1}.SEQ_n = s_z.SEQ_n$ and $certification(t, s_z.SEQ_n)$ holds.
- *Case 2*: $t.site = n$. Following Property 5 (and its notation), by the effects of π_{z_1}, $info(t).start = s_{z_1}.info_n(t).start$. By Property 4.6 and Property 4.9, $info(t).start = s_{z_1}.Ver_n$. As $\neg certification\,(\langle t, info(t)\rangle, s_z.SEQ_n)$ then $\exists \langle t', info(t')\rangle \in s_z.SEQ_n$ such that $info(t').end > info(t).start$ and $info(t').WS \cap info(t).WS \neq \emptyset$. By Property 7, $\exists z_5 \leq z : \pi_{z_5} = end\_commit_n(t') \wedge info(t') = s_{z_5}.info_n(t')$. By the effects of π_{z_5}, $info(t').end = s_{z_5}.Ver_n$. Thus, $s_{z_1}.Ver_n < s_{z_5}.Ver_n \leq s_z.Ver_n$ and obviously, $z_1 < z_5 \leq z$.
  We consider the associated trace of DBS_n to α, i.e. $\beta_n(\alpha)$. We also consider $\beta_n(\alpha[z]) = \beta_n(\alpha)[j]$. In $\beta_n(\alpha)[j]$ taking into account $z_1 < z_5 \leq z$, there exits $i < k \leq j$ such that $\nu_i = \mathbf{begin}_n(t)$, $\nu_k = \mathbf{commit}_n(t')$ and $active_n(t, j)$.
  By Property 5, Property 6, and Property 4.7, $info(t).WS = s_{z_2}.info_n(t).WS = WS_n(t, j)$ holds. By Property 7, Property 4.10 and Definition 4, $info(t').WS = s_{z_5}.info_n(t').WS = WS_n(t', k)$. Thus, $WS_n(t, j) \cap WS_n(t', k) \neq \emptyset$. Assumption 3 (First-Updater-Wins) is violated by $\beta_n(\alpha)$. A contradiction is given.

□

A.7 Proof of Property 13

Proof. The proof is made by contradiction. Assume the property does not hold; i.e. $s_z.status_n(t) \neq active$. By Property 5 (following its notation), $\pi_{z_3} = \mathbf{request\_commit}_n(t)$. As $s_{z_3}.status_n(t) = active$ and $s_{z_3}.sent_n(t) = \texttt{true}$, then $s_z.status_n(t) \notin \{idle, active, blocked, committed\}$. In conclusion, $s_z.status_n(t) = aborted$. The only possible event making $s_z.status_n(t) = aborted$ at site $n \in N$ ($t.site = n$) is $\pi_{z_5} = \mathbf{execute\_ws}_n(t')$ for some $t' \in T$. In addition, $z_1 < z_5 < z$. At s_{z_5-1}, the state in which π_{z_5} is enabled, $s_{z_5-1}.status_n(t) = active$ (it can not be $idle$, $blocked$, $committed$ nor $aborted$ by the fact $s_{z_3}.status_n(t) = active$) and $s_{z_5-1}.info_n(t).WS \cap info(t').WS \neq \emptyset$ where $\langle t', info(t')\rangle = head(s_{z_5-1}.channel_n)$. As $t \in getConflicts(\langle t', info(t')\rangle)$ then $s_{z_5}.status_n(t) = aborted$.
Notice that $\langle t', info(t')\rangle \neq head(s_z.channel_n) = \langle t, info(t)\rangle$. This is only possible if $\pi_{z_6} = \mathbf{end\_commit}_n(t')$ with $z_3 < z_5 < z_6 \leq z$.
By the effects of π_{z_6}, $\langle t', s_{z_6}.info_n(t')\rangle \in s_{z_6}.SEQ_n$, $s_{z_6}.info_n(t').end = s_{z_6}.Ver_n$, and by Properties 6 and 7 $s_{z_6}.info_n(t').WS = info(t').WS$.

It is also simple to show that $s_{z_1}.info_n(t).start = s_{z_1}.Ver_n < s_{z_6}.Ver_n = s_{z_6}.info_n(t').end$.
Again by Properties 5 and 6, $info(t).start = s_{z_1}.info_n(t).start$ and $info(t).WS = s_{z_5-1}.info_n(t).WS$.
Therefore, $\exists\langle t', s_{z_6}.info_n(t')\rangle \in s_z.SEQ_n : s_{z_6}.info_n(t').end > info(t).start \wedge s_{z_6}.info_n(t').WS \cap info(t).start \neq \emptyset$. In conclusion, $\neg certification(\langle t, info(t)\rangle, s_z.SEQ_n)$. A contradiction is obtained.

$\square$